A chemistry-inspired middleware for flexible execution of service based applications

Chen Wang

Thèse

THÈSE INSA Rennes
sous le sceau de l'Université Européenne de Bretagne
pour obtenir le grade de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Informatique

présentée par
Chen Wang
ÉCOLE DOCTORALE : MATISSE
LABORATOIRE : IRISA – UMR6074

A Chemistry-Inspired Middleware for Flexible Execution of Service Based Applications

Thèse soutenue le 28 mai, 2013
devant le jury composé de :

Christophe Cérin
Professeur à l'Université Paris XIII / *Président*

Zsolt Németh
Chercheur au SZTAKI, Hungarian Academy of Sciences / *Rapporteur*

Christian Perez
Directeur de recherche à INRIA / *Rapporteur*

Maurizio Giordano
Chercheur au CNR (National Research Council, Italy) / *Examinateur*

Cédric Tedeschi
Maître de conférence à l'Université de Rennes 1 / *Examinateur*

Jean-Louis Pazat
Professeur à l'INSA de Rennes / *Directeur de thèse*

Dedicated to my parents, Liya Shi and Kening Wang.

Acknowledgments

The best and worst moments of my doctoral journey have been shared with many people. The list of the people I need to thank will not fit to a single Acknowledgments section. I just mention some people whose contribution is obvious.

First and foremost, I want to thank my supervisor Jean-Louis Pazat. I really appreciate all your contributions of time, ideas, and experiences to guide my Ph.D. research. Talking and working with you is really productive and stimulating, which lead to some publications in the top level conferences such as *SCC* and *CCGrid*. Thank you also for giving me so much flexibility in doing my research work, so that my contributions are not only restricted to chemical-inspired approaches but also include some generic solutions in service computing.

In the following, I would like to thank the committee members of my Ph.D. defense. Christophe Cérin, thank you for being the president of the committee to preside the entire process of my Ph.D. defense. Zsolt Németh and Christian Perez, thank you for taking your time to read and to evaluate my thesis manuscript. Your comments are really useful and valuable for me to improve the quality of this dissertation. Cédric Tedeschi and Maurizio Giordano, thank you for participating in the committee. As real experts on chemical computing, you have given interesting questions and valuable suggestions, which provided a clear guide to succeeding researchers who might work on the chemistry-inspired computing in the future.

Part of the work in this thesis is conducted in collaboration with Maurizio Giordano and Claudia Di Napoli, from CNR (National Research Council), Italy. This collaboration started from a discussion with Claudia during the *CIT* conference in the summer of 2010. It was a pleasant talk and we have made the plan for this collaboration. Later, Maurizio spent two weeks in Rennes, France. We have talked a lot about chemical computing and we have also developed together a prototype which finally lead to a publication in *ServiceWave* conference. I would like to thank both of you for this excellent work. It was really a nice experience to work with you.

I have also to thank Thierry Priol, who introduced me to the chemistry-inspired computing and service computing. I still remember the first time you picked me up at the reception desk of INRIA Rennes in November 2008. After a nice conversation, I decided to start a master internship with you for the upcoming 8 months on the topic of using chemical computing to express service orchestration. This subject was totally new for me at that time. And after this internship, thank you for providing me an opportunity to continue this work by starting my Ph.D. research.

Meanwhile, so many thanks to the "chemists" in our research team. Cédric, I want to thank you again for your support and help during the past three years. You gave me a lot of suggestions on my work and my presentations, which helps people to understand my work more easily. Hector, as two of the pioneers working on exploring the use of chemical programming in service computing, we have spent together the most challenging times. I

want to thank you for your support to my work and implementations. Marko, I hope that it is not boring to share an office with me for three years. Thanks for your comments on my work and proofreading my papers before submissions.

All the members of *Myriads* research team in INRIA Rennes also deserve my sincerest thanks, including the ones who have already left during the past three years. Their friendship and assistance has meant more to me than I could ever express. Especially, I have to thank Anca, Stephania and Djawida for the wonderful flyer for my Ph.D. defense.

In addition, I have to thank the large Chinese community, that is the Chinese students, researchers and employees living in Rennes. I feel so lucky to know everyone of you. Since France is far away from China where I was born and then grow up, the huge differences in culture, language and habit sometimes make the life hard. Having different kinds of activities with you (e.g. cooking traditional Chinese food, playing Chinese music, etc.) makes me feel that I was living not that far from home. Especially, I have to thank my Chinese colleagues working in INRIA Rennes, my best friends, Zhaoguang Wang, Ke Sun and Lei Yu. You have made my life more colorful. Thank you.

Last but not the least, I would like to thank my family. First, I have to thank my parents, my mother Liya Shi and my father Kening Wang. Thank you to give birth to me and your endless love brings me inspiration and courage, which is always my driving force. Finally, I also have to strongly thank my girl friend (fiancee now), Jiayi Liu, a Ph.D. student in Telecom Bretagne, France. I would like to say, I could not finish my Ph.D. thesis without your support and love. Love you forever!

Chen Wang
Myriads Research Team - INRIA
June 2013, France

Contents

List of Figures

List of Tables

List of HOCL Programs

Introduction

1 Research Context: Service-Oriented Computing

With the continuous progress in global economy and technology, enterprises have changed the way to do business. In the past, enterprises developed separately desktop (or Web-based) applications to manage their business. Nowadays, the boom in global market economy has greatly promoted the collaboration among enterprises. For example, with the emergence of Electronic Business (E-Business) [150], more and more enterprises have developed Web-based platforms that enable customers to learn, to select and to purchase their products on line. The development of such platforms requires enterprises to implement a variety of software modules providing different services/functionalities, such as online payment (money transfer) services, express delivery services. However, all these software modules can be hardly implemented and provided by a single enterprise. Therefore, today's business information systems call for new paradigms for designing distributed applications that span organizational boundaries and heterogenous computing platforms.

Service-Oriented Architecture (SOA) [66] is widely adopted today by many enterprises as a flexible solution for building loosely coupled distributed applications. From the viewpoint of SOA, each software module can be provided as an internet-accessible *service*, defined as a self-contained, self-describing, autonomous and platform-independent software component that can receive requests and return computational results through a set of well-defined and standard interfaces. Recently, the advent of cloud computing promotes such Software-as-a-Service (SaaS) model [21]. Running over cloud infrastructures, services are provided as public utilities, such as electricity. Following the *pay-as-you-go* model, a service requester is charged based on his/her consumptions of a service, for example, the number of requests to this service.

Service-Oriented Computing (SOC) paradigm utilizes services as fundamental building blocks for developing applications [118]. Using SOC approaches, a distributed (business) application can be easily developed by defining a *service composition*, represented by a *workflow*, which integrates and coordinates a set of *constituent service*s. All constituent services can be provided either internally or externally from the enterprise, running on distributed infrastructures and heterogenous platforms. Compared to traditional approaches, the development of such Service-Based Applications (SBA) can reduce the cost and complexity. On one side, since SBA providers can directly reuse (or purchase) functional modules from other service providers, both labor cost and time consumption in software development can be greatly reduced. On the other side, constituent services are managed independently by their own providers; accordingly, the cost in software maintenance is dispensable for SBA providers.

However, due to the distributed and loosely coupled execution environment, the execution of SBA must be *flexible* in order to achieve both functional and non-functional business goals. Flexible execution of SBA has to meet the following requirements:

1. *Dynamicity*: each SBA instance can be constructed on the fly in response to various requirements of different end users.

2. *Adaptability*: the execution of an SBA instance is required to be adaptable to various runtime changes (e.g. failures).

3. *Autonomicity*: both dynamicity and adaptability have to be realized in an autonomic way to minimize the intrinsic complexity to operators and users.

Obviously, flexible execution of SBA exhibits a high degree of challenges as well as complexities from both academic and practical perspectives.

2 Motivations

Such flexible service-based systems share a high degree of similarities with biological systems due to the dynamic, adaptive and distributed nature. Recently, nature-inspired metaphors have been identified as promising approaches to model such self-adaptable and long-lasting evolving service-based systems [135].

2.1 Nature-Inspired Computing

People have regressively learned from the nature. Throughout human's history, a lot of inventions and ideas have been developed with the inspiration derived from the biological and natural world. For example, two thousand years ago, a famous Chinese carpenter invented the saw with the inspiration from a kind of grass with teeth along its edge[2]. As another example in modern times, the airplane was invented based on the study and the analysis on bird's wings.

From the perspective of the research scientists of computer systems, the nature is the largest distributed system, which presents the best example on how to efficiently and effectively build self-managed and self-adaptive distributed systems. The Nature-Inspired Computing (NIC) [94] aims at developing computational models and algorithms inspired from natural metaphors, including physics, chemistry, and biology, to solve practical and complex problems in large-scale distributed systems.

Chemistry-Inspired Computing. Chemical programming model [29] is a research branch of nature-inspired computing. It is an innovative programming paradigm for parallel and autonomic computing. Inspired by the metaphor from chemistry, a program is described as a chemical solution, where all the floated molecules represent computational elements (e.g. data). The computation is modeled as a series of reactions controlled by a set of reaction rules. As the laws of the computation, a reaction rule specifies the consumption and the production of molecules: it can modify/consume the existing molecules (the reactant) or generate the new ones (the resultant). The computation completes when the solution becomes *inert*, defined as the state where no more reaction can be triggered.

[2]The story can be found at `http://history.cultural-china.com/en/38History6890.html`

2.2 Objectives

This dissertation aims to investigate novel approaches in response to the challenges in building flexible service-based systems. Some preliminary research work have been conducted to demonstrate the feasibility and viability in using the chemical programming model to program autonomous service-based systems [32, 31]. The *main objective* of this dissertation is to extend these work by designing, developing and evaluating a chemistry-inspired service middleware for flexible execution of SBAs. The middleware is expected to deal with some of the most important steps in SBA's runtime management:

The construction of SBAs: how to flexibly select constituent services? Using the Internet as the medium, more and more services become available which can provide the same functionality under different Quality of Service (QoS) (e.g. cost, response time, etc.). In response to various requirements from different end users, an SBA is required to be dynamically constructed by selecting and integrating the "most suitable" constituent services on the fly.

The execution of SBAs: how to flexibly coordinate constituent services? Orchestration and choreography represent respectively the centralized and decentralized model for service coordination [122]. Compared to the orchestration model, choreography can improve the performance in scalability, throughput and execution time; whereas, it also brings additional complexities and challenges, such as fault-tolerance, security and privacy issues. Therefore, future-generation service-based system is required to be able to flexibly select the "best" model to execute a service composition according to its execution context.

The adaptation of SBAs: how to flexibly react to runtime failures? A running SBA instance may fail due to the distributed, heterogenous and loosely coupled execution environment. For example, network congestion can lead to a constituent service completely responseless, which will in succession cause the failure in executing the corresponding SBA instance. In this context, the execution of SBA is required to be self-adaptable on the fly in order to react to various failures raised from functional and non-functional levels.

Additionally, on the way to build flexible service based systems, we do not restrict our research only in investigating the chemical-based solutions. In this context, the *second objective* of this thesis is to find out generic solutions, such as models and algorithms, to respond to some of the most challenging problems in flexible execution of SBAs. For example, how to proactively predict and avoid the occurrence of failures instead of passively reacting to them?

3 Contributions

To achieve these objectives, the main contributions of this dissertation are two-fold:

A chemistry-inspired middleware for flexible execution of SBAs. I have proposed and developed a biochemistry-inspired service middleware that models service-based systems as distributed, self-organized and self-adaptive biochemical systems. 1) Firstly, all the service-related concepts (e.g. services, data, etc.) are described using biochemical metaphor, such as atoms, molecules and cells; 2) then, by using an illustrative example

in the real life, we have demonstrated flexible execution of SBA in terms of a series of pervasive chemical reactions controlled by a number of rules. The contributions of this work cover the follows aspects.

1. **Flexible selection of services[3].** We have defined a number of *instantiation rules* that describe service selection as a series of molecular polymerization processes. Both local and global constraints can be easily expressed using reaction rules. Different from other traditional approaches that model service selection as a sequential or a one-shot process, in the middleware, the services are selected in a recursive and aggregation way: chemical reactions can be executed independently and concurrently.

2. **Flexible coordination of services.** By providing different groups of *coordination rule*s, we have realized a centralized *orchestration* model as well as two decentralized models to coordinate constituent services, namely *semi-choreography* and *auto-choreography*. An SBA provider is able to specify a desirable model to execute service compositions based on his/her execution context. Furthermore, a number of experiments have been conducted by running two experimental workflows in the middleware to compare different models in terms of efficiency and complexity.

3. **Flexible adaptation of services.** We have integrated a number of *adaptation rules* in the middleware that describe reactive adaptation actions as a series of chemical reactions. These reactions can modify the binding references or even the workflow structures in order to cope with runtime failures raised from both functional and non-functional aspects.

Compared to traditional approaches, chemical-based implementation exhibits some desirable characteristics, such as higher-level abstraction, simplicity, evolvability and self-adaptability.

Towards proactive adaptation of SBAs. The chemistry-inspired middleware has realized some of the most important steps in SBA's management life cycle, from its construction, to its execution and adaptation. However, it can only react to rather than prevent the occurrence of failures. In this work, we get a step further to investigate *proactive adaptation* [100]. The objective is to guarantee the end-to-end QoS of SBA by executing preventive adaptation actions before QoS degradations actually occur. Since runtime adaptation is costly, one of the key challenges to efficiently implement proactive adaptation is to accurately draw adaptation decision in order to avoid unnecessary adaptations. I have proposed a *two-phase online prediction approach* capable of accurately forecasting an upcoming degradation in the end-to-end QoS of a running SBA instance as early as possible. This approach is evaluated and validated by a series of realistic simulations. The results have shown that our approach is able to draw both accurate and timely adaptation decision compared to other traditional prediction approaches. This contribution is more generic, which can not only be integrated into the chemistry-inspired middleware but also implemented by other traditional approaches.

[3]This work is conducted in collaboration with Claudia Di Napoli and Maurizio Giordano from CNR (National Research Council, Italy)

4 Publications

International Conferences

1. Chen Wang and Jean-Louis Pazat: A Chemistry-Inspired Middleware for Self-Adaptive Service Orchestration and Choreography. In *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'2013)*. In Delft, The Netherlands (May 13–16, 2013).

2. Chen Wang and Jean-Louis Pazat: A Two-Phase Online Prediction Approach for Accurate and Timely Adaptation Decision. In *the 9th IEEE International Conference on Service Computing (SCC'2012)*. Honolulu, Hawaii, USA (June 2012).

3. Claudia Di Napoli, Maurizio Giordano, Jean-Louis Pazat and Chen Wang: A Chemical Based Middleware for Workflow Instantiation and Execution. In *the 3rd European Conference on ServiceWave (ServiceWave'2010)*: 100-111. Gent, Belgium (December 2010).

4. Chen Wang and Jean-Louis Pazat: Using Chemical Metaphor to Express Workflow and Service Orchestration. In *the 10th IEEE International Conference on Computer and Information Technology (CIT'2010)*: 1504-1511. Bradford, UK (June 2010).

National Conferences

1. Chen Wang and Jean-Louis Pazat: Un middleware inspiré par la chimie pour l'exécution et l'adaptation flexible des applications basées sur services. In *RenPar'21 (Rencontres francophones du Parallélisme)*. Grenoble, France (January 2013).

Tutorials & Technical Reports

1. Chen Wang: A QoS-Aware Middleware for Dynamic and Adaptive Service Execution. (May 2011) Available online at HAL-INRIA: *http://hal.inria.fr/hal-00794027/*.

2. Chen Wang: A Middleware Based on Chemical Computing for Service Execution - Current Problems and Solutions. (Jan. 2011) Available online at HAL-INRIA: *http://hal.inria.fr/hal-00794023/*.

3. Chen Wang, Thierry Priol: HOCL Programming Guide. Technique report (Sept. 2009). Available online in HAL-INRIA: *http://hal.inria.fr/hal-00705283/*.

4. Chen Wang, Thierry Priol: HOCL Installation Guide. Technique report (Aug. 2009). Available online at HAL-INRIA: *http://hal.inria.fr/hal-00794028/*.

5 Organization of Dissertation

This dissertation is organized in three parts.

Part I: Background

In this part, we provide the context of this work.

- **Chapter 1** starts with a general introduction to service-oriented computing. Some cutting-edge research problems and the related state-of-the-art research work are discussed.

- **Chapter 2** first investigates some unconventional approaches for programming service-based systems, i.e., rule-based systems and tuple-space based systems. Then, we present a similar but more preferable approach known as chemical programming model. Finally, we introduce an implementation of chemical programming model named as Higher-Order Chemical Language (HOCL).

Part II: Chemistry-Inspired Middleware

This part focuses on the first contribution of our work. We will present the design, the implementation and the evaluation of a middleware based on chemical programming model for flexible execution of service-based applications.

- **Chapter 3** focuses on the design of the middleware. First, the architectural overview of the middleware is illustrated. Then, we present a number of reaction rules that describe service selection in terms of a series of chemical reactions in the middleware. In the following, a centralized model (*orchestration*) and two decentralized models (*semi-choreography* and *auto-choreography*) are introduced for autonomous service coordination and adaptation.
- **Chapter 4** focuses on the implementation and the evaluation of the middleware. First of all, we analyze the performance of different models by using two experimental workflows. Then, we present the implementation of HOCL and the distributed chemical infrastructures on which the middleware is running. Finally, as a proof-of-concept validation to show its viability, a number of experiments have been conducted by executing two experimental workflows in the middleware. The evaluation results prove our analysis in the beginning of this section.

Part III: Towards Proactive Adaptation of SBA

In this part, we present the second contribution of our work.

- **Chapter 5** discusses how to determine the best timing to proactively start adaptation before failures actually occur. A *two-phase online prediction* approach is introduced to forecast an upcoming failure as early as possible while keeping a higher prediction accuracy. This approach is evaluated based on a series of realistic simulations using different kinds of workflows.

Conclusions and Perspectives

Finally, we conclude this work with the discussion on the future work.

Bibliography

In this part, a list of sources used in this dissertation are provided.

Appendix

The appendix provides some additional information (examples and source codes) to support this dissertation.

Part I

backgroud

Chapter 1

Service-Oriented Computing

Abstract. In this chapter, we introduce the background of this dissertation. First of all, some important concepts in service-oriented computing are presented in Section 1.1. In the following, some major research problems in service computing are discussed and the related state-or-the-art research work are presented. Section 1.2 addresses the construction of service-based applications by solving the service selection problem. Both local and global service selection approaches are presented. Then, in Section 1.3, we introduce both centralized and decentralized models and their implementation (tools/languages) to execute service-based applications. Finally, Section 1.4 discusses runtime adaptation of service-based applications in order to react to runtime changes in the execution environments, such as failures. Different types of adaptation techniques are introduced.

1.1 Service-Based Application

Service-Oriented Architecture (SOA) provides a new perspective to build rapid, low-cost, interoperable and evolvable distributed applications. From the perspective of SOA, any piece of code, application component or even software system can be transformed into a network-available *service*. With the popularity of the Internet, a service can be consumed remotely through a set of well-defined interfaces by means of message exchanges. In this context, the development of a distributed application is performed by creating a Service-Based Application (SBA) that assembles and coordinates a number of existing services available via the Internet.

Figure 1.1 illustrates an example of SBA. Enterprise A intends to develop a service-based application in order to provide a certain functionality to is clients. First, an *abstract workflow* is defined by decomposing the expected functionality into a collection of interrelated *tasks* (or *activities*) that function in a logical order to achieve the ultimate business goal. Such an abstract workflow is not executable because it lacks the binding reference for each task. Therefore, the abstract workflow is required to be instantiated before the execution can start. The instantiation of workflow aims to construct an executable *concrete workflow* by mapping each task t_i to a specific service, noted as $ws(t_i)$, as addressed later in Section 1.2.

A concrete workflow presents a *service composition* which outsources the execution of each task t_i to a specific service $ws(t_i)$. In this context, all the services involved in a service composition are defined as *constituent service*s (e.g., service S_1, S_2 and S_3 in Figure 1.1). All the constituent services can be developed based on different technologies, provided by different enterprises/organizations and running on heterogenous platforms. The execution of a service composition is performed by a series of interactions to all the constituent services. Different models to coordinate the interactions among constituent services will be introduced in section 1.3.

The advent of cloud computing promotes such Software-as-a-Service (SaaS) model. Following the concept of "*pay-as-you-go*", the consumer/requester of a service only pays a little amount of money to the service provider for each invocation (some service providers may offer service packages which allow unlimited invocations within a certain period). Accordingly, compared to traditional approaches, the service-oriented paradigm enables the SBA provider (e.g. enterprise A) to reduce the cost and the complexity in designing and developing complex applications from scratch. Additionally, the service-based application exhibits the following characteristics:

- **Distributed resources.** The service composition represents distributed computational resources, from both software level and hardware level.
- **Heterogeneity.** Services can be implemented in different programming languages, and running on heterogenous platforms or system.
- **Cross administration domain.** Services can be developed and managed by different organizations/enterprises.
- **Loose coupling.** A service is a self-contained functional unit; thus, the modification of a service implementation has minimum effect on others (clients).

In this section, we introduce some important concepts in service-oriented computing, which are the basis of our discussions in the rest of this dissertation.

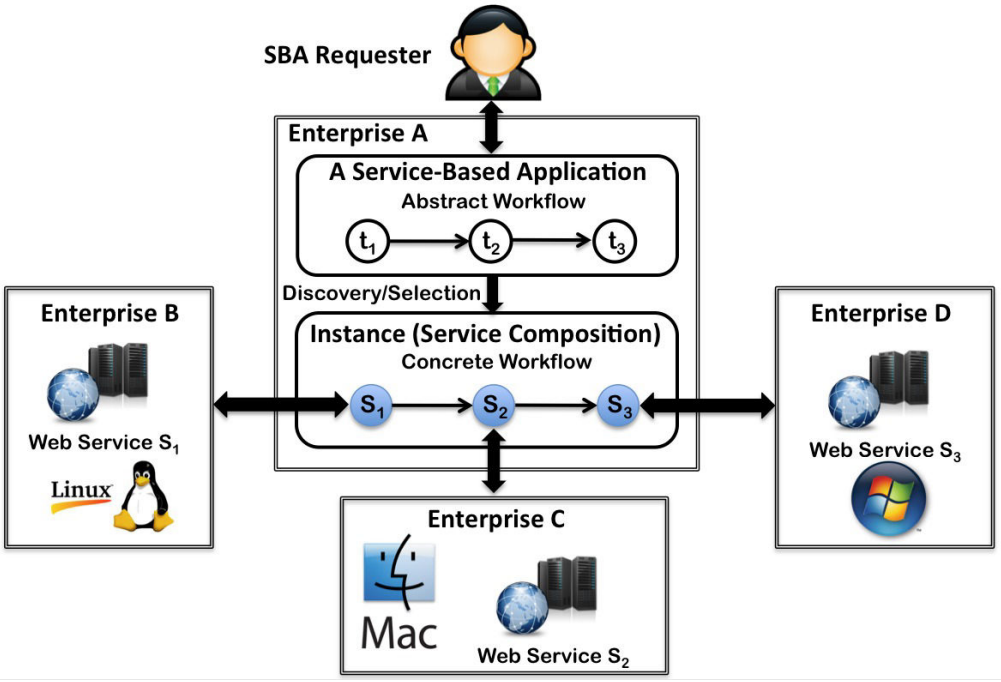


Figure 1.1: Service-Based Application

1.1.1 Web Service

Web service is the basic building block for developing Service-Based Applications (SBA). From the above discussion, we can see that a Web service acts as a black box that encapsulates software and hardware resources. It can be consumed (or reused) by exchanging messages through a set of program interfaces, which specify the message formats and communications protocols. However, it is hard to find an exact definition for the term *Web service* since many of them exist. In this dissertation, we use the definition provided by W3C[1]:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

From the above description, Web service refers to a software system that is reusable remotely over an Internet protocol backbone, to name a few, Extensible Markup Language (XML) [1] for presenting data, Simple Object Access Protocol (SOAP) [4] for transmitting data, the Web Service Description Language (WSDL) [6] for describing service interfaces etc. The interface is implemented by a concrete software *agent*, which can be a concrete piece of software or hardware that realizes the expected functionalities.

Web Service Architecture. As shown in Figure 1.2, the Web Services architecture consists of three roles: service provider, service requester and service registry.

[1]W3C (the World Wide Web Consortium) is an international community that develops open standards to ensure the long-term growth of the Web. `http://www.w3.org/TR/ws-arch/`

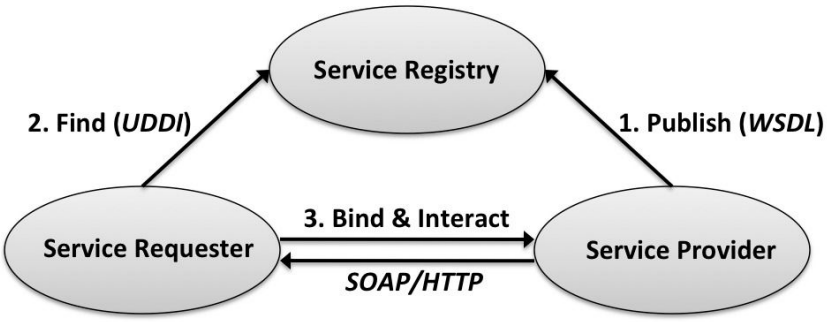


Figure 1.2: Web Service Architecture

- Service provider is the a person (or an organization) that has implemented a Web service to provide certain computational functionalities through a set of well-defined interfaces.
- Service requester is a person (or an organization) that consumes existing web service(s) by means of exchanges message exchanges.
- Service registry is a directory of Web services, where service providers can publish new services and requesters can find existing ones.

In order to make Web service architecture work, three fundamental operations have been defined: *publish*, *find*, and *bind*.

- First of all, service providers have to publish their services to a service registry/broker by advertising both functional interfaces as well as non-functional properties of its Web services.
- In the following, a service requester needs to go to the the registry to find the Web service(s) capable of providing the expected functionality. The registry will return the corresponding endpoint references of the candidate Web services.
- Finally, the requester binds and interacts with all the selected Web service(s) in order to benefit the expected functionality.

Atomic Web service VS composite Web service. According to different implementations, a service can be classified into two categories: an *atomic service* or a *composite service*. The former represents an individual software component that provides the expected functionality; whereas the latter defines a service composition which aggregates a number of basic and fine-grained services. An atomic service is an indivisible self-contained piece of software. By contrast, the composite service first decomposes the realization of the expected functionality into several inter-related tasks, and then outsources the execution of each task to an Internet-accessible Web service. Figure 1.3 describes the hierarchy of service composition: a composite service can aggregate both atomic and other composite Web service(s). Please notice that either an atomic or a composite Web service describes a service from the perspective of its implementation. However, from the viewpoint of a service requester, any service is atomic because it can only know the interface of a Web service but has no knowledge on how a Web service is implemented.

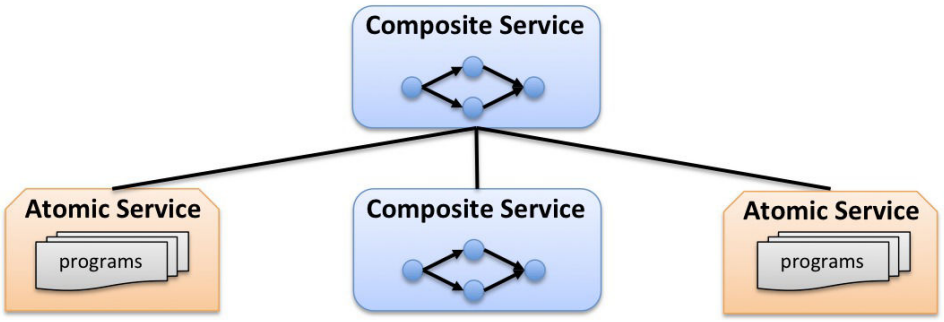


Figure 1.3: Hierarchy of Service Compositions

1.1.2 Workflow

A workflow describes the business logic by specifying the execution order of tasks (services). The topology of a workflow is represented by a graph where the nodes represent abstract tasks and the edges specify the dependencies between these tasks (services). Any workflow can be transformed to a Directed Acyclic Graph (DAG) by removing loops. Two approaches are proposed in the literature. 1) [158] introduces an unfolding method: all the tasks between the beginning and the end of a loop are cloned K times, where K is the maximum number of times that this loop has been executed according to the past execution logs. 2) In [20], loops peeling is introduced as an improvement of unfolding technique: loop iterations are represented as a sequence of branching evaluations. Each evaluation aims to decide whether to continue or to exit the loop. Therefore, the loop structure will not be considered in this dissertation.

In order to express complex control flow, some special nodes have to be added to the DAG workflow representation describing complex workflow patterns [134]. As an example, Figure 1.4 shows a concrete workflow which integrates and coordinates 10 Web services. First of all, an *and-split* workflow pattern is added after S_2 since the execution will be diverged into two parallel branches when S_2 is executed. Then, a *synchronization* workflow pattern is located before S_9 because all the execution branches will be synchronized before the execution of service S_9. Similarly, after the execution of service S_3, each outgoing branch is associated with a condition and the execution will continue to only one of them according to the outcome of the service S_3. In this case, an *exclusive-split* is required after S_3 and a *simple-merge* is used before S_5. More description about complex workflow patterns can be found in [8].

In the following, we introduce two important concepts, namely the Execution Plan (EPL) and the *Execution Path* (EPA). An execution path is a sequential execution of tasks from the first task to the last one. An execution plan is a possible execution of workflow which contains all the tasks in parallel branches and all the tasks in only one of exclusive branches; Let P and Q indicate respectively the number of execution plans and paths in a workflow, npl_p and npa_q indicate respectively the number of the tasks included in the execution plan epl_p and execution path epa_q. The workflow shown in Figure 1.4 has three execution paths (Q=3):

- $epa_1 = \{S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_6 \rightarrow, S_9 \rightarrow S_{10}\}$; $(npa_1 = 7)$
- $epa_2 = \{S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_5 \rightarrow S_6 \rightarrow S_9 \rightarrow S_{10}\}$; $(npa_2 = 7)$
- $epa_3 = \{S_1 \rightarrow S_2 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10}\}$; $(npa_3 = 6)$

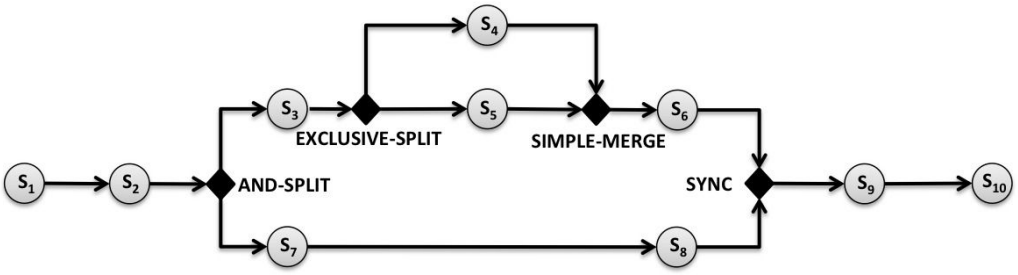


Figure 1.4: Sample Workflow

and two possible execution plans (P=2):

- $epl_1 = \{S_1, S_2, S_3, S_4, S_6, S_7, S_8, S_9, S_{10}\}$; $(npl_1 = 9)$
- $epl_2 = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\}$; $(npl_1 = 9)$

The execution plan epl_1 (epl_2) includes all the tasks in the execution paths epa_1 and epa_3 (epa_2 and epa_3). Let $prob_p$ is the probability to execute the execution plan epl_p, we have $\sum_{p=1}^{p=P} prob_p = 1$. In this example, we assume that $prob_1 = 0.3$ and $prob_2 = 0.7$.

1.1.3 Quality of Service

The Quality of Service (QoS) reflects the non-functional performance of a Web service. A QoS attribute indicates how well a Web service performs in terms of a specific quality metrics. We discuss only five of them in this dissertation. More descriptions of the QoS attributes can be found in [12].

- **Cost**. The cost $q_c(ws)$ is the fee that a service requester has to pay in order to invoke Web service ws. Its value can be any non-negative float numbers.
- **Time**. The response time $q_t(ws)$ reflects the expected duration for delivering the expected functionality by Web service ws. It is a positive float number measured in seconds.
- **Availability**. The availability $q_{av}(ws)$ reflects the probability that the Web service ws stays deliverable. Therefore, its value is a real number from 0 to 1.
- **Security**. The security $q_{sec}(ws)$ reflects the ability of the Web service ws to provide authentication, authorization, confidentiality and data encryption. It has a set of enumerated values, such as HIGH, MEDIUM, LOW.
- **Reputation**. The reputation $q_{rep}(ws)$ is used to measure the trustworthiness of the Web service ws. Its value is normally calculated based on the feedback of all the requesters in the past according to their experiences in purchasing/invoking ws. Its value normally varies within a range of numbers, for example, a real number from 0 to 5.

A QoS attribute can be either *numeric* or *descriptive*. The quality of a numeric attribute can be expressed by a number while the descriptive attribute uses an expression to describe the quality levels. Considering the five QoS attributes mentioned above, the cost, response time, availability and reputation are numeric QoS attributes whereas the

Table 1.1: QoS Aggregation Function

QoS	QoS Aggregation Function	Global QoS Calculation
q_c	$aqos_c(epl_p) = \sum_{t_i \in epl_p, t_i \leftarrow ws_i} (q_c(ws_i))$	$gqos_c = max\{aqos_c(epl_p), 1 \leq p \leq P\}$
q_t	$aqos_t(epa_q) = \sum_{t_i \in epa_q, t_i \leftarrow ws_i} (q_t(ws_i))$	$gqos_t = max\{aqos_t(epa_q), 1 \leq q \leq Q\}$
q_{av}	$aqos_{av}(epl_p) = \prod_{t_i \in epl_p, t_i \leftarrow ws_i} (q_{av}(ws_i))$	$gqos_{av} = \sum_{p=1}^{P} prob_p \cdot aqos_{av}(epl_p)$
q_{sec}	$aqos_{sec}(epl_p) = min\{q_{sec(ws_i)} \mid t_i \in epl_p, t_i \leftarrow ws_i\}$	$gqos_{sec} = min\{q_{sec}(ws_i) \mid t_i \leftarrow ws_i\}$
q_{rep}	$aqos_{rep}(epl_p) = \frac{1}{npl_p} \cdot \sum_{t_i \in epl_p, t_i \leftarrow ws_i} (q_{rep}(ws_i))$	$gqos_{rep} = \sum_{p=1}^{P} prob_p \cdot aqos_{rep}(epl_p)$

security is a descriptive one. Moreover, a numeric QoS attribute can be either *positive* or *negative*. For positive attributes, the higher value results in a higher quality, such as availability and reputation; by contrast, price and execution time are negative attributes since the greater value leads a lower quality.

QoS Aggregation. The end-to-end QoS of SBA is determined by the QoS of all component Web services. For example, the execution time of a service composition depends on how fast each constituent Web service responds. Before presenting how to calculate the end-to-end QoS of SBA, we first introduce three concepts.

- **Local QoS (lqos).** The local QoS refers to the QoS of each constituent Web service.
- **Aggregated QoS (aqos).** The aggregated QoS reflects the end-to-end QoS of an execution of SBA (an execution plan).
- **Global QoS (gqos).** The global QoS reflect the end-to-end quality level of an SBA by considering all the possible execution plans.

Table 1.1 shows how to calculate the aggregated QoS and the global QoS based on all the local ones. Different QoS attributes are aggregated using different aggregation functions. 1) the aggregated cost of an execution plan epl_p is the sum of the cost of all the constituent services that are included in epl_p. 2) The execution time is aggregated along each execution path epa_q. Its value is the sum of the response time of all constituent services that are included in epa_q. 3) An execution plan epl_p is available if and only if all the constituent services are available. In this case, the aggregated availability is the product of the availabilities of all the constituent services in epl_p. 4) The aggregated security level depends on the lowest security level of all Web services in an execution plan epl_p. 5) The aggregated reputation is calculated by the average rating of all the constituent services in an execution plan.

The global QoS reflects the expected end-to-end QoS for an execution of SBA, expressed as $gqos = (gqos_c, gqos_t, gqos_{av}, gqos_{sec}, gqos_{rep})$ The global price and time are determined by the worst execution case: the aggregated price and execution time cannot exceed these limits for all possible execution plans. Similarly, the global security only depends on the constituent service that implements the lowest security level. By contrast, the availability and the reputation is calculated based on the historical information over a long periods. The values of these attributes generally reflects all execution possibilities (the possibilities of different execution plans have to be considered).

Table 1.2: Local QoS

QoS	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
price (€)	0.10	0.05	0.05	0.00	0.02	0.05	0.08	0.03	0.20	0.00
time (s)	10	15	12	15	5	8	15	15	25	5
availability	0.998	0.996	0.993	0.987	0.999	0.992	0.993	0.990	0.997	1.00
security	+++	+++	+++	+++	++	+++	+++	+++	+++	+++
Reputation	5	5	5	4	5	3	4	5	5	5

Table 1.3: Aggregated QoS

QoS	$path_1$ (epa_1)	$path_2$ (epa_2)	$path_3$ (epa_3)	$plan_1$ (epl_1) - 30%	$plan_2$ (epl_2) - 70%
price (€)	-	-	-	0.56	0.58
time (s)	90	80	85	-	-
availability	-	-	-	0.947	0.959
security	-	-	-	+++	++
Reputation	-	-	-	4.222	4.667

Example. In the following, we use an example to demonstrate the computation of the aggregated and the global QoS of SBA. Take the workflow defined in Figure 1.4 for example, the local QoS of all the constituent service are provided in Table 1.2. Firstly, we use the aggregation function defined in Table 1.1 to compute the aggregated QoS. We only demonstrate the computation of the aggregated execution time as a proof-of-concept example. The execution time is aggregated along each execution path. Thus, the aggregated execution time of the path epa_1 is simply computed as follows:

$$aqos_t(epa_1)=q_t(S_1)+q_t(S_2)+q_t(S_3)+q_t(S_4)+q_t(S_6)+q_t(S_9)+q_t(S_{10})=90 \text{ seconds.}$$

Similarly, the aggregated execution time of the other two execution paths are respectively 80 and 85 seconds. The aggregated values of other QoS attributes are calculated in the similar way, which is provided in Table 1.3.

Based on the aggregated QoS, we are able to compute the global QoS of SBA. By using the global QoS calculation function defined in Table 1.1, we can see that the global price, execution time and security is determined by the worst case. In our example, the global price, execution time and security level are respectively 0.58 €, 90 seconds and medium (++). Then, the availability and reputation takes the probability into consideration. As we have assumed, the probability to execute epl_1 and epl_2 are respectively 30% and 70%. Thus, the global availability and reputation equal to 0.955 and 4.535 respectively. The global QoS can be therefore expressed as: $gqos = (0.58, 90, 0.955, \text{MEDIUM}, 4.535)$.

1.1.4 Service Level Agreement

A Service Level Agreement (SLA) is a mutually-agreed contract between the service requester and provider, which dictates the expectations as well as obligations in regards to how a service is expected to be provided. On one hand, the expected quality level is formulated by specifying agreed target values for a collection of QoS attributes. On the other hand, some penalty measures are defined in case of failing to meet these quality expectations. The definition of penalty is beyond the discussion of this thesis. Accordingly, our discussion will focus on the definition of quality level of an SLA.

An SLA groups a set of relevant QoS attributes to specify a quality level of a Web service which states how "well" the service is expected to respond. For the reason of

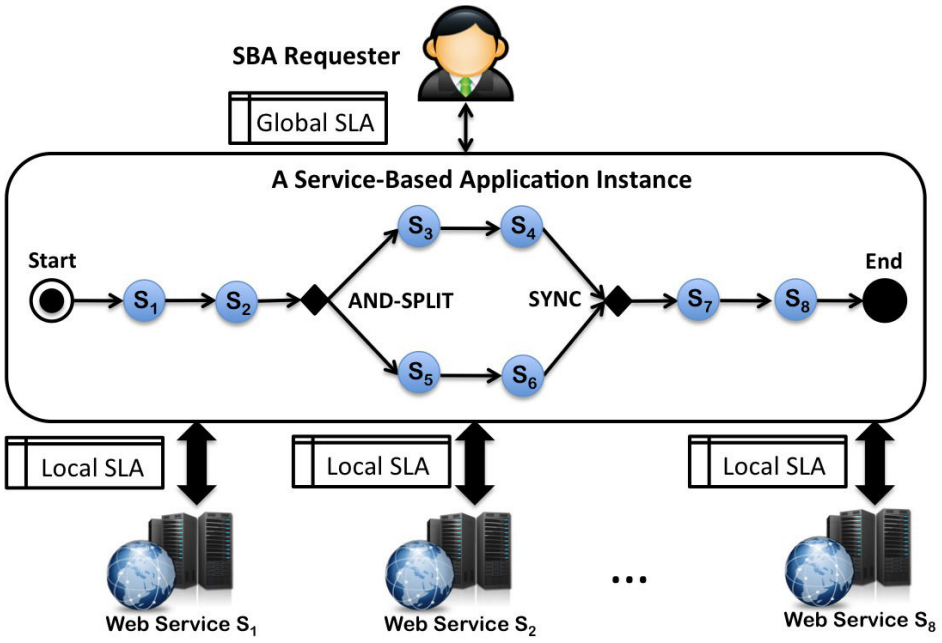


Figure 1.5: Local and Global SLAs

simplicity, in this dissertation, an SLA is modeled as a set of target values of QoS attributes. As an example, a typical definition of SLA can be described as follows:

> Each invocation to Web service ws costs 0.1 €, and the Web service ws is expected to respond within 5 seconds, with an availability of 0.99.

The above SLA can be expressed as: $sla(ws) =< q_t = 2, q_c = 0.1, q_{av} = 0.99 >$.

Local SLA and Global SLA. In the context of service-based application, the SBA provider plays the roles of service provider as well as consumer: it consumes a group of constituent Web services in order to provide value-added service to its customers. For both roles, it establishes an SLA with each of its counterparts, as shown in Figure 1.5. The SLA between the SBA and a constituent Web service is defined as *local SLA*, and the one negotiated with the end requester is defined as *global SLA*. A local SLA reflects the expected QoS of a specific constituent service to execute a workflow task, denoted as $lsla(S_i) =< q_t(S_i), q_c(S_i), ... >$, where $q_t(S_i)$ and $q_c(S_i)$ represent respectively the expected time consumption and cost for invoking service S_i. More QoS attributes can be used here. Whereas the global SLA dictates the end-to-end performance of the entire SBA, denoted as $gsla =< gqos_t, gqos_c, ... >$, where $gqos_t$ and $gqos_c$ specify respectively the expected end-to-end time consumption and the overall cost for an execution of SBA.

Obviously, the definition of the global SLA depends on the local ones. As an example, Table 1.4 lists the expected execution time and the cost defined in both global and local SLAs for the SBA instance depicted in Figure 1.5. For each QoS attribute, the value of global QoS can be executed by using the aggregation function introduced in Section 1.1.3. In this example, the global execution time and global cost for for the service composition are respectively 6,400 s and 1.4 €. As a result, the SBA provider promises that it is capable to respond with 6,800 s (with 400 s for the cost of coordination and safety margin) at the cost of 1.8 €per invocation (it earns 0.4 €per invocation).

Table 1.4: Example of Global and Local SLAs

QoS Attributes	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	SBA
time (s)	600	800	1500	1500	1800	1000	1200	1000	6800
price (€)	0.1	0.3	0.05	0.0	0.15	0.5	0.1	0.2	1.8

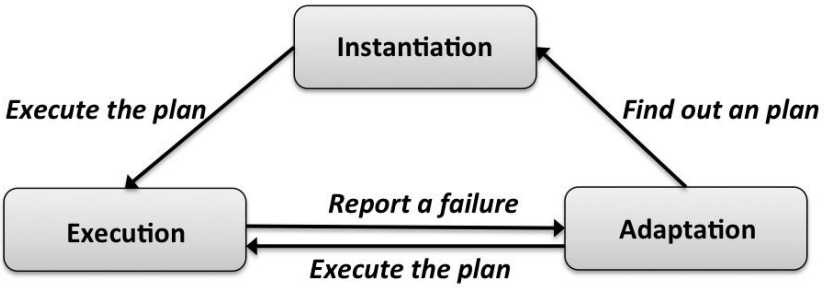


Figure 1.6: Life cycle of SBA Runtime Management

1.1.5 Runtime Management of Service-Based Applications

Due to the dynamic and distributed execution environment, the runtime management of a running service-based application exhibits a high degree of complexities and challenges. As shown in Figure 1.6, the life cycle of SBA runtime management can be generally divided into three phases.

Instantiation. The instantiation phase aims to build a concrete workflow by discovering, selecting and integrating suitable constituent services at runtime.

Execution. The execution phase is to run an instantiated SBA instance by coordinating all constituent services.

Adaptation. The objective of the adaptation phase is to react to runtime failures in order to guarantee both functional and non-functional expectations in delivering the SBA.

All these phases are inter-related. Firstly, the instantiation phase is the basis of the execution of SBA. Then, adaptation phase accepts the failures reported from the execution phase and identifies suitable adaptation plans to cope with these failures. Finally, sometime, the adaptation plan requires the re-instantiation of (a part of) workflow. In the rest of this section, we introduce a number of the cutting-edge research problems related to these three phases and some state-of-art solutions.

1.2 Instantiation of Workflow

Different SBA requesters may have various non-functional requirements, for example, the maximum cost that (s)he can afford or the expected response time. In response to different requests, the SBA is always abstractly defined at the design time by composing a number of inter-dependent tasks as place holders. When a request arrives, this abstract workflow is instantiated by selecting a suitable constituent service for each task. The instantiation

of workflow aims to construct a concrete workflow that can best fit for the requester's requirements. It requires several functional steps to fulfill the instantiation process.

- **Advertise.** First, each service provider has to publish its Web service by describing both functional and non-functional properties in a publicly available registry so that other enterprises are able to find it and then consume it.
- **Discover.** The registry provides the discover function so that a service requester is able to find the services with his/her expected functionality.
- **Select.** If the requester has found multiple functional-equivalent services, (s)he has to select the best one(s) based on different criteria.
- **Bind.** Finally, each workflow task has to bind to the selected service in order to outsource the computation for the required functionality.

Since the advertise and discover functions are always addressed by the service registry (refer to Figure 1.2), our discussion focuses on the implementation of the selection and bind functions. Compared to the bind function, service selection is more complicated and challenging. Using Internet as the medium, more and more Web services are available today. Therefore, it is possible to have multiple functional-equivalent candidate Web services to execute a certain task. With the widespread proliferation of Web services, quality of service (QoS) becomes a significant factor in distinguishing functional-equivalent Web services. A *service class* is defined as a set of Web services that can provide the same functionality but differ in quality of services. The service selection process aims to select exact one Web service from the service class of each task in order to construct a concrete workflow. Given an abstract workflow with N tasks, denoted as $awf=\{t_i|1 \leq i \leq N\}$. The relative service class for each task t_i is defined as $SC_i=\{S_{i,j}|1 \leq j \leq M_i\}$, M_i is the number of services in service class SC_i. The objective of instantiation is to construct a concrete workflow $cwf=\{t_i \rightarrow S_{i,k}|1 \leq i \leq N, 1 \leq k \leq M_i\}$ by mapping each abstract task t_i to a selected Web service $S_{i,k}$. In this section, different selection approaches are introduced.

1.2.1 Static Selection v.s. Dynamic Selection.

The selection and integration of constituent services can be performed either at design time or at runtime. The design-time service selection refers to the *static service selection* that reflects provider-centric business model. In this context, an SBA provider selects different sets of services to predefine multiple concrete workflows at design time, which can execute the SBA on different quality levels. The requester can select a specific quality level to execute the SBA that best fits for his/her nonfunctional requirements (preferences or constraints).

For example, the SBA provider pre-defines two concrete workflows: cwf_1 and cwf_2. cwf_1 integrates high quality (e.g. fast response time) but costly Web service for each task, whereas cws_2 uses low-cost or even free web services which have limited and unassured non-functional qualities. Therefore, the SBA can be provided on two levels: higher quality level (but expensive) by executing cwf_1 and lower quality level (but cheap) by executing cwf_2. Then the requester can purchase a suitable level to execute the SBA according to his/her budget.

By contrast, runtime service selection presents client-centric business model. In this context, a requester explicitly expresses his/her QoS constraints and preferences, and all

the constituent services are selected at runtime so that the global QoS of entire service composition can satisfy the end requester's QoS requirements. Compared to static service selection, dynamic selection exhibits a high degree of dynamicity in customizing the quality quality level of SBA.

1.2.2 Local Selection v.s. Global Selection.

Dynamic service selection introduces the optimization problem which can be solved either locally or globally. The local approaches aim to select the best candidate for each task separately. Accordingly, it is more efficient but the requester's constraints on the end-to-end QoS cannot be ensured.

1.2.2.1 Local Service Selection

In [47], the eFlow system is proposed that allows nodes (tasks) to have service selection rules. When the eFlow engine tries to execute an activity it calls a service broker that executes the service selection rules and returns a list of candidate services (with ranking information). [104] introduces an approach for QoS specification and service selection. The selection algorithm takes into account both the trustworthiness of a service provider and the relative benefit offered by a provider with respect to the requester-specified QoS criteria. In [79], the authors present an architecture for dynamic Web service selection within a workflow enactment process. The dynamic selection of services is performed through a Proxy Service interacting with a Discovery Service and an Optimization Service.

In [158], the authors present AgFlow: a middleware platform that enables QoS driven service composition. Two alternative service selection approaches are discussed: local optimization and global planning. The local approach implements a greedy algorithm that selects the best candidate for each activity individually; The global approach selects the services by separately optimizing each execution path. As demonstrated in [19], this global approach cannot always guarantee the global QoS. In [13], the authors presented a hybrid approach that combines global optimization with local selection. The idea is to firstly decompose global QoS constraints into a set of local ones by using Mixed Integer Programing (MIP), and then the best candidate is selected locally for each activity independently. When QoS decomposition results in a set of restrictive local constraints, the local selection may fail. Hence, a service composition cannot be always successfully identified even though it actually exists.

1.2.2.2 Global Service Selection

By contrast, the global selection approaches aim to identify a service composition that can meet the requester's end-to-end QoS requirements. In [19], the authors prove that global service selection is equivalent to a Multiple-choice Multi-dimensional Knapsack Problem (MMKP) [73], which is proved to be NP-Hard. As a result, identifying the best service composition often results in a higher time complexity. The solution can be generally classified by two categories: 1) the service selection is formulated as an optimization model and an optimizer (such as CPLEX [3]) is used to compute the optimal solution. 2) some heuristic algorithms are proposed to find a near optimal (sub-optimal) solution in less time.

Optimization approaches. In [19], the service selection is modeled as a Mixed Integer Linear Programming (MILP) problem where both local and global QoS constraints can

be specified. Their work is extended in [20] by introducing loop peeling and negotiation technique. The experimental results shows that it is effective for large processes even with severe constraints. [154] proposes two optimization models for solving service selection problem: *combinatorial model* and *graph model*. Using combinatorial model, service selection is modeled as a Multiple-choice Multiple-dimension Knapsack Problem (MMKP); And the graph model formulates the selection problem as Multi-Constrained Optimal Path (MCOP) problem [87]. But the combinational model is only suitable for linear workflows.

Heuristic approaches. As argued in [14], linear programming is not suitable for runtime service selection since the time complexity to obtain the optimal solution is exponential. In addition, the objective of service selection is to find out a feasible service composition rather than the best one that can satisfy the end requester's requirements. Accordingly, other approaches propose heuristic algorithms to efficiently find a near-optimal solution. In [155], the authors extend both models they proposed in [154] to the general workflow case and introduce a heuristic algorithm for each model to find near-optimal solutions in polynomial time, which is more suitable for real-time service selections. In [98], the authors present a heuristic algorithm based on clustering techniques, which exhibits satisfying efficiency in terms of time cost and optimality.

1.3 Service Interaction Models

After the instantiation phase, a concrete service composition is constructed and the SBA is ready for the execution. The execution of a service composition is performed by coordinating all the constituent services, by means of message exchange, so that they are able to collaborate in order to achieve the ultimate business goal. Orchestration and choreography are two perspectives to model service interactions in executing a service composition [122].

- **Web Service Orchestration.** First, service orchestration presents the perspective of centralized coordination model. By implementing an executable business process (e.g. WS-BPEL [5]), the business logic and execution order are expressed from a single party's viewpoint. The business process is executed by a centralized execution engine which coordinates a series of invocations to all constituent services.
- **Web Service Choreography.** By contrast, service choreography refers to decentralized and cooperative service coordination. From the perspective of service choreography, the centralized coordination is eliminated and the coordination is distributed among all constituent services. As a result, each constituent service, acting as a peer, can interact directly with each other. The execution of a service composition is thus performed by peer-to-peer collaborations among all participants.

Different coordination models exhibit different degrees of complexity and efficiency. In this section, we are going to present some state-of-the-art implementations of both centralized and decentralized coordination models and discuss their advantages as well as the limitations .

1.3.1 Web Service Orchestration.

Service orchestration refers to the centralized coordination of constituent services. The most common approach to implement a Web service orchestration is to use an orchestration language (e.g. WS-BPEL [5]) to define an executable business process. It can be

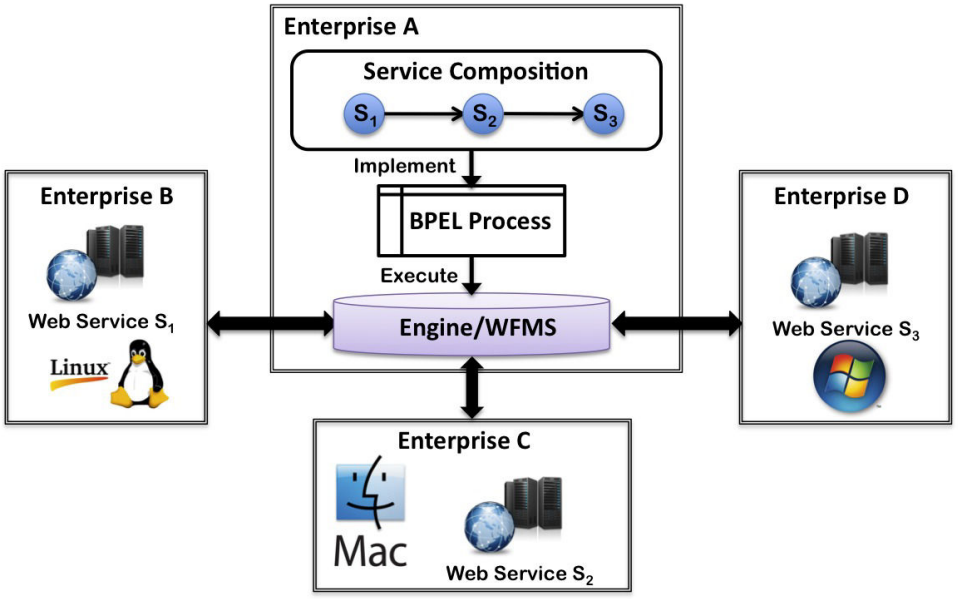


Figure 1.7: Web Service Orchestration Model

executed by a centralized engine, e.g., a Workflow Management System (WFMS), which coordinates a series of invocations to all constituent services, as shown in Figure 1.7. From the perspective of the service orchestration, a constituent service has no knowledge about how a service composition is defined. As a result, each constituent service is not aware of the existence of other participants. For example, the Web service S_2, S_3 and S_4 in Figure 1.7 do not know each other. They only communicate with the centralized coordinator, namely the business process implemented by enterprise A. In the following, we are going to present several platforms for describing and executing Web service orchestration.

WS-BPEL Engine. WS-BPEL (Business Process Execution Language for Web Services) [5] is the most widely used orchestration language. As announced by OASIS (Organization for the Advancement of Structured Information Standards) [2], the development of WS-BPEL aims at enabling users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks.

Using BPEL, a business processes can be described in both abstract and executable ways. An abstract process refers to an abstract workflow that lacks concrete operational details. It can serve a descriptive role, with more than one possible use case, including observable behavior and process template. Whereas an executable business processes represents a concrete workflow, which can be executed by a BPEL engine to actually coordinate the constituent services in business interactions. When a BPEL process receives a request from the end requester, a new instance is created. Each instance starts with the *receive* operation that receives the invocation data from the end requester, and terminates with a *reply* activity that returns the final computing results to the end requester. In between, a number of activities are defined and connected in order to describe the workflow.

Activities can be divided into 2 classes: basic and structured. A *basic activity* expresses an elemental functional unit of a business process. For example, the *invoke* activity calls a Web service with the required data and the *assign* activity initiates or assigns a value to a

[2]OASIS is a not-for-profit consortium that drives the development, convergence and adoption of open standards for the global information society. https://www.oasis-open.org/

variable. A running BPEL instance also supports fault handling mechanisms, a *compensate* activity is used to undo a part of past execution in case of encountering runtime errors. By contrast, a structured activity can be seen as a block of constructs, or a container, which prescribes the relationship of several activities (can be both basic and structured). For example, the *sequence* and *flow* activity describes respectively the sequential and the parallel execution of a set of activities. The *if* activity expresses the conditional semantics of the execution and the *while* activity defines a loop of execution. A *scope* activity divides the BPEL program into several blocks.

Then, the BPEL file is executed by a BPEL engine, which interprets the description of workflow and executes the flow chart of tasks by coordinating a series of invocations to all constituent services. A variety of implementations of BPEL engine exist. To name a few, ActiveVOS [9], Apache ODE [17], Oracle BPEL Process Manager [114], IBM WebSphere Process Server [80] and etc. An overview and a comparison of different BPEL engines can be found in `http://en.wikipedia.org/wiki/Comparison_of_BPEL_engines`.

Workflow Management Systems. One of the advantage of BPEL is that it is proposed by a standardization committee and supported by many key IT industry players. However, a number of alternative workflow description language exists, which are implemented by some workflow management systems. In the following, we are going to present several popular workflow management systems which can be used to describe and run Web service orchestrations.

YAWL (Yet Another Workflow Language) [152] is an orchestration language based on high-level Petri net. It supports multiple instances, composite tasks, complex workflow patterns, etc. Using YAWL, a workflow is described as a set of extended workflow nets which form a hierarchy architecture: atomic tasks is represented by the leaves and a composite task refers to a sub EWF-net at a lower level. The YAWL engine is a fully open-source workflow system which provides a graphical editor for developers to design workflow. With the built-in verification functionality, the developers are able to detect potentially errors. It also has open interfaces based on Web standards, which enable developers to plug-in existing applications and to extend and customize the system in many ways.

Kepler [85] is a free software system for designing, executing, reusing, evolving, archiving, and sharing scientific workflows. Kepler builds upon the Ptolemy II framework, developed at the University of California, Berkeley. Kepler workflows can be exchanged in XML using Ptolemy's Modeling Markup Language (MoML). The workflow is defined by a composition of actors, which are connected through the *port*s.

SCUFL (Simplified Conceptual United Flow Language) [132] is a data-flow oriented workflow description language. Using SCUFL, each execution step is described as a processor that represents a web service or another executable application component. A processor receives raw data from its input(s), processes the data and generates the results to its output(s). Therefore, the workflow is described as a set of pipes that link all the processors. SCUFL is used by Taverna [131], an open source domain independent Workflow Management System for designing and executing scientific workflows, such as the applications for bioinformatics, chemoinformatics, astronomy and etc.

The Pegasus Workflow Management System [121] executes workflow-based applications on heterogenous environments including desktops, campus clusters, grids, and clouds. The workflow is described as a directed acyclic graph composed of tasks and data dependencies between them. It can map abstract and high-level workflow description, especially scientific workflows, onto a set of distributed available computing resources, perform optimizations,

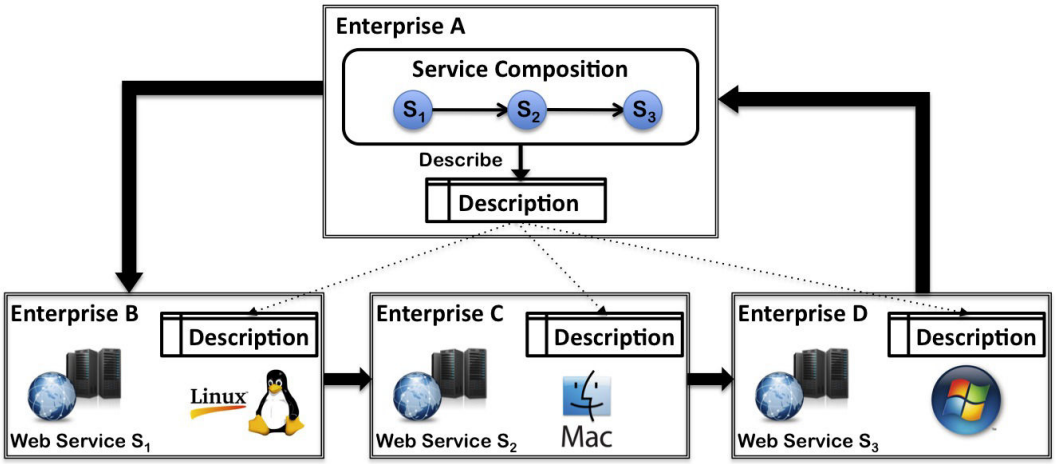


Figure 1.8: Web Service Choreography Model

and reliably execute the steps in the appropriate order. This makes workflows portable by enabling the user to define the workflow once, and run it anywhere.

More detailed presentation and comparison of current implementations of service orchestration engine and WFMS can be found in [67].

1.3.2 Web Service Choreography.

The simplicity is the most important advantage of the orchestration model. First, each constituent service only needs to interact with the centralized coordinator, from whom it receives the invocation message. It is not required to do any coordination task for an execution of SBA. Secondly, the centralized engine facilitates the runtime management of the execution of service composition, such as the reactions to failures. The coordinator has the global view on the execution state of workflow, thus it is capable to make appropriate decision when failures arise. Finally, the centralized control exhibits better security. Some runtime decisions depend on the internal business policies of an enterprise (e.g. selection among exclusive branches). Accordingly, the provider does not want to distribute such policies to other constituent services for decentralized coordination.

However, the centralized control may result in performance degradation. 1) Since all the messages have to go back to the centralized execution engine each time rather than delivered directed between constituent services, the overall execution time may increase. 2) The centralized execution engine or WFMS introduces the bottleneck of the service-based system, which may lead to huge network traffic, especially when big data exchange is required. 3) The centralized orchestrator becomes a potential Single Point of Failure (SPF). It can become unavailable (e.g. responseless) in case of overload, especially for large-scale service composition with complex workflow.

In contrast to the orchestration model, Web service choreography is proposed for decentralized coordination of Web services. The word *choreography* is defined as:

> Dancers dance following a global scenario without a single point of control.[3],

Web service choreography models the execution of a service composition as a dance, each constituent Web service acts as a dancer, which does a part of computational task according to the behaviors of the other participants. As a result, unlike the service orches-

[3]The definition is from Wikipedia [7], the free encyclopedia.

tration model which requires a centralized orchestrator to coordinate constituent services, choreography model represents the autonomous and collaborative cooperation among all constituent services.

As shown in Figure 1.8, a workflow is described collaboratively by all constituent services. A description of workflow specifies the role that each service plays in a service composition and their relationship. Each constituent service keeps a copy of workflow description so that they are aware of the global data and control flow. For example, when Web service S_1 completes the computation, instead of returning the result to the SBA, it forwards directly the computational result to Web service S_2. In the following, service S_2 will start the computation immediately and then forwards the result directly to S_3. As a result, by interpreting the global description of workflow, each participant knows 1) when to start the computation, 2) its job in a service composition and 3) how to forward the results.

Choreography Languages. The Web Services Choreography Description Language (WS-CDL) [136] is an XML-based language that describes peer-to-peer collaborations of Web Services by defining, from a global viewpoint, their common and complementary observable behavior. *Let's Dance* [157] is a visual choreography language targeted at business analysts. It does not allow any technology-specific configurations. However, interface behavior descriptions out of the global interaction model can be generated. *BPEL4Chor* [58] is the choreography extension for BPEL. The extensions facilitate a seamless integration between service choreographies and orchestrations. Here, choreographies serve as starting point for generating participant behavior descriptions for each service which are then used for implementing new services or for adapting existing services. Vice versa, bottom-up approaches, where existing BPEL processes are interconnected, are helpful for analyzing the overall interaction behavior between services and optimizing it. Since BPEL is an accepted standard and has a defined execution semantics, we use it as foundation for describing choreographies. as WS-CDL comes with its own set of control flow constructs that can hardly be mapped to those of BPEL. Due to the limited space, the details of the other choreography languages, such as *ScriptOrc* [38], *Multiagent Protocols (MAP)* [35], are not provided here.

1.3.3 Decentralized Service Orchestration.

Limitations of the orchestration and the choreography models. Compared to the orchestration model, service choreography can improve the performance in throughput, scalability and response time due to the cooperative interaction and collaborative coordination. On one side, data can be passed directly from the source to the destination rather than travel by the centralized coordination point. Therefore, the execution of a service composition is more efficient and fast, especially when big data exchange is required. On the other side, by removing the centralized orchestrator, the performance bottleneck and SPF has been eliminated, which improves the scalability and robustness of the system.

However, the implementation of service choreography presents a high degree of complexity and challenge in the context of service collaboration across enterprise boundaries.

Design-time complexity. Firstly, the greatest challenges result from the development of a choreography of services. All constituent services are developed and managed independently by different organizations across heterogeneous platforms. Therefore, the direct communication between them may lead to some runtime problems, for example, the interface mismatch. In this scenario, each service provider is required to modify his/her

Web service (interface) in accordance with other service providers. However, it brings additional cost and complexity. Moreover, it is unpractical for the provider of each constituent service since the modification of the interface of its service will lead to the failures in serving with other clients.

An alternative solution is to develop a special copy of service with the same implementation, the mediated interfaces and the coordination information (e.g. each constituent service has to know its preceding and succeeding services). However, this solution is neither unpractical. On one side, in the context of on-demand execution of SBA, a service composition is identified at runtime during the instantiation phase. Thus it is challenging to generate a set of connectable copies of constituent services at runtime. On the other side, a choreography of service presents a high level of dynamicity. For example, for another execution instance of SBA in Figure 1.8, a different service composition is identified, which may not include service S_2 any more. Therefore, it is costly to generate a copy of service for only one-shot usage.

Runtime complexity. Furthermore, without a centralized controller, the runtime management of service choreography is challenging. For example, in this distributed and loosely coupled execution environment, a constituent service may become responseless due to the failures from infrastructural level. Such a responseless constituent service may not be noticed by the other participants, which will lead to the execution of the entire service composition incompletable.

Security and privacy. Finally decentralized coordination can lead to privacy and security issues. As stated before, each participant is required to do a part of coordination task, the SBA provider is required to distribute some policies (e.g. decision algorithms) to other enterprises. Such information may include highly private business information, which is not possible to be distributed to other organizations.

The needs for decentralized orchestration. The orchestration model introduces performance bottleneck and scalability problem, whereas the choreography model is complicated to manage due to both design-time and runtime challenges. Thus, some research work propose decentralized orchestration model, which takes the advantages from both models. As illustrated in Figure 1.9, the idea is to break the workflow description file (e.g. BPEL file) into several smaller pieces. All pieces of code, also defined as *orchestration fragment*s are distributed on a number of inter-connected physical machines. Each machine is configured with an orchestration engine that is able to interpret the orchestration fragment and to execute a part of service composition. By this means, the centralized point of coordination is replaced by a group of coordinators, which can increase the parallelism and improves the performance in terms of throughput, response time and availability. Moreover, all the machines are managed by the SBA provider (e.g., enterprise A in Figure 1.9), which can easily resolve the problems raised by cross-organizational execution environment.

Implementation of decentralized orchestration. [50] requires each constituent Web service to implement a business process engine in order to execute a part of code. However, [120] argues the limitation of such "pure" choreography model in the context of the collaboration across administrative domains. The authors present the FOCAS framework which executes the fragments of a service composition on a set of internal inter-connected orchestration engines. Similarly, in [91], the authors introduce a distributed agent-based orchestration engine in which each agent is able to execute a portion of business process and collaborate with others. The fault handling issue for decentralized coordination is

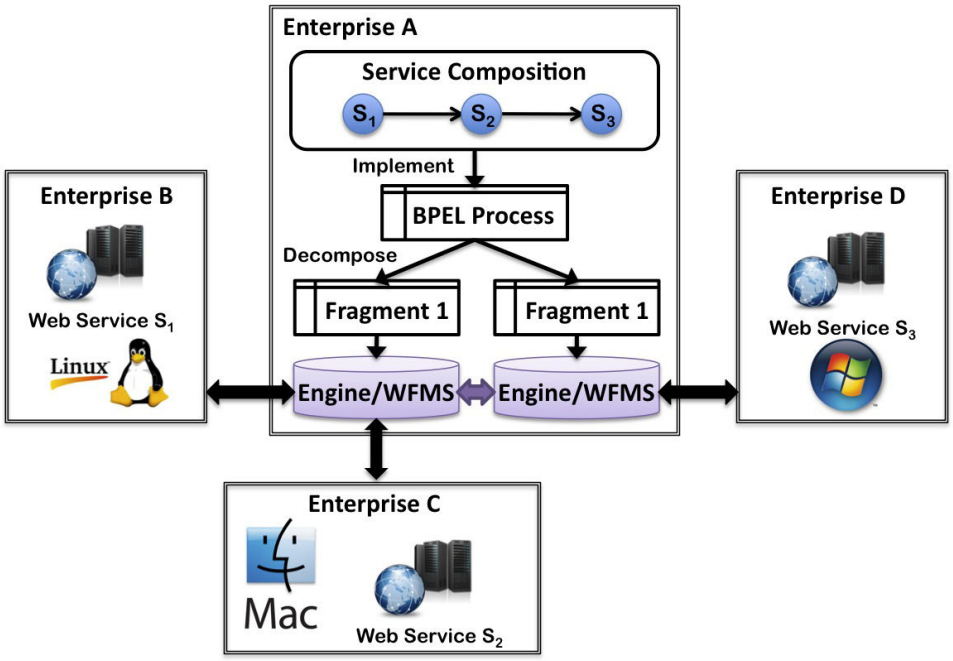


Figure 1.9: Decentralized Service Orchestration Model

addressed in [48]. By placing fault handlers in each partition, a failure can be captured and propagated to the corresponding handler.

1.4 Runtime Adaptation of Service-Based Applications

Due to the loosely coupled execution environment, the execution of SBA may fail, or fail to meet the required quality level. For example, a constituent service may take longer time to respond due to the network congestion; moreover, infrastructure failures can cause a service completely responseless. Therefore, the execution of SBA is required to be adaptable to both functional and non-functional failures [44, 49]. The generic solution for building self-adaptive service system is to implement the MAPE control-feedback loop (Monitor-Analyze-Plan-Execute) [84], which is comprised of four *functions* (as shown in Figure 1.10).

- **Monitor.** Firstly, the execution each SBA instance is monitored by intercepting communication messages in order to collect a series of events;
- **Analyze.** The events generated by the *monitor* function are used to evaluate the quality state of a running SBA instance and to analyze the need for adaptation;
- **Plan.** Once an adaptation decision is determined, a suitable adaptation plan (e.g. a list of adaptation actions) is identified;
- **Execution**[4]**.** Finally the relative countermeasures of an adaptation plan are applied to this running SBA instance.

[4]The term *execution* here refers to the execution of an adaptation plan rather than the execution of the workflow.

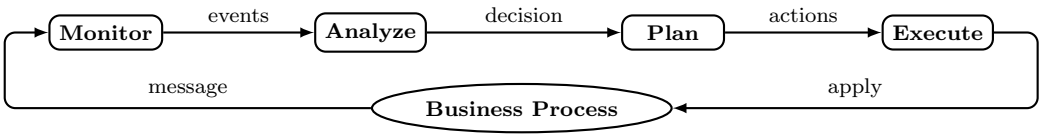


Figure 1.10: MAPE Control-Feedback Loop

Based on the different implementations of the MAPE loop, all the approaches for runtime adaptation of SBA can be classified into two categories with different objectives. 1) Firstly, *reactive adaptation* aims to recover the execution from runtime failures by executing some remedial countermeasures. The objective is to cope with failures so as to guarantee the completion of an execution of SBA. 2) By contrast, *preventive adaptation* approaches rely on the prediction of an upcoming failure in the future so that the adaptation actions can be executed before failures actually occur. The objective is to prevent the occurrence of failures. In this section, we are going to introduce some state-of-the-art work for both reactive and preventive adaptations.

1.4.1 Reactive Adaptation

The reactive adaptation aims to build a self-healing service-based systems[123]: adaptation plans will be executed once a runtime failure is detected (based on the *monitor* function). In the following, we first introduce different types of failures and then present the corresponding adaptation plans.

1.4.1.1 Runtime Failures

Generally speaking, a failure can be classified into two categories, *functional* failure and *non-functional* failure.

Functional (Infrastructural) level failures. A functional failure refers to a runtime error that will lead to the execution of SBA undeliverable, which means that the execution completes with error or even cannot complete. The cause of a functional failure can be manifold. In this dissertation, our discussion only focuses on the failures raised from the infrastructural level. For example, an invocation/response message can be lost due to the network problem. Or the server where a constituent service is hosted may become responseless due to too many invocations.

Non-functional (Quality-of-Service) level failures. As we have introduced in Section 1.1.4, an SBA provider may establish a global SLA with its clients. In this case, the QoS level failure refers to degradation of the end-to-end QoS that leads to the violation of the global SLA. For example, an SBA provider promises to reply within an expected duration. However, since some of the constituent services may respond slowly, the execution of this SBA instance may take longer time than expected.

1.4.1.2 Reactive Adaptation Plans

In the context of reactive adaptation, the *analyze* function is implemented as an *event-based* trigger. Once a certain type of failure is reported (by the *monitor* function), a suitable adaptation plan is then selected and executed. In the following, we present several

adaptation plans to cope with both functional and non-functional level failures introduced above.

Change Binding reference. The change of a binding reference refers to the substitution of a constituent service by another functional-equivalent one. It always aims to respond to functional level failures, for example, the recovery from the failure in invoking a constituent service.

Some research work aim to extend WS-BPEL (engine) to support self-adaptive BPEL process [101, 102, 93, 34]. WS-BPEL provides fault handlers as the interfaces to define adaptation plans. [130] presents a self-healing BPEL process engine named SH-BPEL engine. By building plug-ins to the existing BPEL engines, the authors propose different levels of recovery mechanisms. [101] extends BPEL with annotations. BPEL designers can specify recovery plans as annotations at the design time. And then the annotated BPEL will be translated to a standard BPEL that can be interpreted by any BPEL engine. By using this approach, no change in BPEL execution engine is required for realizing recovery actions. In [93], the authors propose a declarative approach to specify fault-handling adaptation plan by a set of Event- Condition-Action (ECA) rules. These ECA rules are then integrated with normal business logic to generate the fault-tolerant BPEL process.

Some systems are developed as add-ons to the existing BPEL framework/engine in order to provide reactive adaptability. In [18], the framework PAWS is developed for flexible and adaptive execution of service-based processes. But it requires SBA providers to specify both local and global QoS constraints at designing phase. These local constraints are used as criteria to select, bind and replace partner services at runtime. In [103], the authors introduce VieDame, a system that can monitor BPEL processes by intercepting SOAP messages, and then select/replace partner services at runtime based on various strategies. However, the endpoints of all candidate services are stored in a local repository.

Recompose workflow. Workflow recomposition refers to recompose (a part of) the rest of workflow[5] by re-selecting and re-binding each unexecuted task to a new functional-equivalent constituent service with better expected QoS. The objective of workflow recomposition is to improve the end-to-end quality of SBA when a QoS failure arises.

Some research work model service adaptation as a re-instantiation process [158, 19, 20]. Therefore, most of service selection approaches presented in Section 1.2.2 can be directly reused. In [75], the authors investigate adaptation of service composition based on workflow patterns. An adaptation plan is identified according to the execution logic between the failed task and its neighbors. For different workflow patterns, the adaptation can be execution either by replacing a constituent service or recomposing a part of workflow. The cost for identifying and executing adaptation plan is also considered.

Modify workflow structure. Sometime, the adaptation is required to modify the workflow structure. For example, the execution of a task fails but no alternative one is found. SBA provider may substitute the failed task by another workflow fragments that can provide the same functionality. Or sometime, SBA provider has to modify the control flow or data flow due to the change of policy. Moreover, it is also possible to optimize the global QoS by modifying the workflow structure (a possible solution for QoS level failures). To the best of our knowledge, due to a high level of complexity, runtime modification of workflow structure is not widely studied. A similar work can be found in [82].

[5]Here, the rest of workflow refers to the part of workflow tasks that are left unexecuted yet.

1.4.2 Preventive Adaptation

Different from reactive adaptation, preventive adaptation aims to execute the corresponding adaptation plan once a failure is predicted (rather than occurred). Therefore, all the adaptation plans introduced above can be also used for the preventive adaptation to prevent the occurrence of failures. The key challenge to implement preventive adaptation is to determine the need for adaptation in order to draw accurate adaptation decisions (the *analyze* function). All existing approaches to implement the *analyze* function can be classified into two categories.

- **Offline analysis.** Offline approaches can decide when and how to proactively execute adaptation plans by reasoning the causes of the past failures. The adaptation aims at preventing failures for the future executions rather than the ongoing ones.
- **Online prediction (proactive adaptation).** By contrast, online approaches is able to prevent the failures for each running SBA instance by predicting an upcoming failure. And thereby, preventive adaptation is proactively execute in order to avoid the occurrences of failures.

1.4.2.1 Offline Analysis Approaches.

The offline approaches reason global SLA violations on the composition model level and cannot prevent SLA violation for each running instance. In [40], the authors present MoDe4SLA (Monitoring Dependencies for SLAs) approach to analyze SLA dependencies during development phase and then monitor these dependencies at runtime. Based on the past executions, a feedback is provided to SBA providers showing which partner service performs good and which performs bad. Finally, by analyzing the event log of past executions, If a partner service often violates its local SLA, it is then replaced by a functional-equivalent service with better QoS. In this way, SBA provider can replace the service that can hardly meet local SLA with a functional equivalent one with better quality. Later, this work is improved in [41] by introducing formalization of dependency analysis and feedback models. [92] uses structural equation to model the QoS measurement of web services. Then, the change of quality of service can be quantitatively predicted by using prediction mechanism of structural equation model. In [15], the authors argue that most of QoS prediction approaches are based on time series forecasting models that cannot guarantee accurate QoS forecasting where these models are based on a homogeneity (constant variation over time) assumption [42]. To address this limitation, the authors integrates ARIMA and GARCH models in order to capture the QoS attributes' volatility and to provide accurate forecasts.

1.4.2.2 Online Prediction Approaches.

The existing online prediction approaches in the literature can forecast the failures raised by either *functional failure*s or *non-functional deviation*s.

Online testing approaches. Some research work use online testing techniques to test all constituent services in parallel to the execution of an SBA instance. By this means, an upcoming functional failure can be forecasted before its real occurrence. [77] presents the PROSA framework, which defines key activities to initiate online testing either on the binding level or on the service composition level, and thereby proactively triggers the adaptation process. [64] investigates how to guarantee functional correctness of conversational

services. The authors propose a novel approach enabling proactive adaptations through just-in-time testing. Online testing approach is helpful to detect potential functional failures but it can hardly be aware of the deviation of the end-to-end QoS. Furthermore, it requires each consistent service to provide a test mode (e.g. free interfaces for testing).

Runtime verification. Our research work belongs to the latter case, which predicts SLA violation by asserting non-functional deviations that might happen at the end of the execution (e.g. delay). Some research work use runtime verification techniques to determine the necessity of adaptation and to trigger preventive adaptation. In [128], the authors introduce SPADE approach: after the execution of each task, if the local SLA is violated, SPADE uses both monitored data and the assumptions to verify whether the global SLA can be still satisfied. If it reveals that the global SLA is tent to be violated, the adaptation is accordingly triggered. However, early verifications are largely based on the assumptions rather than monitored data, thus they are inaccurate and can lead to many unnecessary adaptations.

Machine learning. Other research work use machine learning techniques in order to provide precise predictions and avoid unnecessary adaptations. In [90], a set of concrete points are defined in the workflow as checkpoints. Each checkpoint is associated with a predictor, which is implemented by a regression classifier. When the execution of the workflow reaches to a checkpoint, the corresponding predictor is activated and uses the knowledge learned from past executions to predict whether the global SLA will be violated. This work is extended in [89] by proposing the PREvent framework, which integrates event-based monitoring, runtime prediction of SLA violations and automated runtime adaptations. Such checkpoint-based prediction approach has some limitations: firstly, some misbehavior (e.g. huge delay) between two checkpoints cannot be handled in time. In the meantime, the best adaptation opportunity might be lost. Furthermore, poorly selected checkpoint(s) may lead to undesirable results, such as unnecessary adaptations. But the selection of optimal checkpoints is complicated and challenging, especially for complex workflows.

Event-Driven Approach Some online prediction approaches are based on processing runtime events. Event-processing aims to autonomously response to the change of execution environment by analyzing and processing streams of information (data) generated (monitored) at runtime. [65] introduces proactive event-driven computing. The authors extend the event processing agent model to include two more type of agents: predictive agents and proactive agents. predictive agents are able to predict a (collection of) possible failures in the future while proactive agents dynamically computes optimal adaptation plans. In [159] presents a novel event-driven QoS prediction system that is able to forecast key performance indicators (KPI) based on a collection of event-analysis techniques.

Chapter 2

Unconventional Approaches for Flexible Service Management

Abstract. Traditional approaches to implement Web service compositions exhibit some limitations, such as inflexibility in expressing service coordination and reacting to runtime changes. In this chapter, we investigate some unconventional approaches that are considered as suitable to program service-based systems due to better flexibility and adaptability. Firstly, in Section 2.1, both rule-based systems and tuple-space based systems are introduced. In the following, our discussion in Section 2.2 focuses on a similar but more preferable approach known as chemical programming model. Some preliminary research work in using chemical-based approaches for expressing and managing service-based applications are presented. Finally, Section 2.3 provides a real-life example of SBA that will be used in the rest part of this dissertation to illustrate our approach.

2.1 Unconventional Paradigms for Service Computing

Traditional approaches to implement a Web service composition rely on the use of executable languages (such as WS-BPEL [5]) to define a business process. The coordination of service is expressed by a number of structured activities in terms of XML-based specifications. Each structured activity acts as a block (or a container), defined by a pair of XML labels, which describes the relationship of all the activities included in between. For example, all the activities defined between <sequence> and </sequence> labels will by executed in sequential order. In this case, a service composition is defined as a monolithic block, which comprises a hierarchy of structured activities.

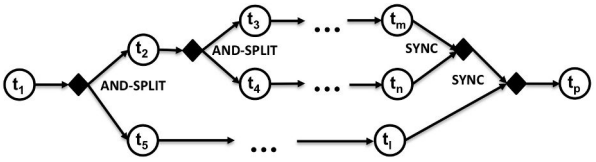
However, these traditional approaches have several limitations. First of all, the definition of a business process is cumbersome, especially for complex workflows which comprise a lot of logical expressions (e.g. parallel and exclusive branches). Take the workflow WF_1 defined in Figure 2.1(a) for example, using traditional approaches, its definition is provided in Figure 2.1(d), which is composed of a number of nested structured activities. More complex workflows will result in a higher level of nesting depth (imagine that the description for a large-scale scientific workflow can be a disaster).

Furthermore, this kind of structured workflow definition is incapable to express unstructured workflow. As analyzed in [95], for a given workflow, if every split workflow pattern (e.g. AND-split, XOR-split, etc.) has exactly one corresponding join workflow pattern (e.g. synchronization, simple-merge, etc), it is defined as a *structured workflow*; otherwise, it is an unstructured workflow. For example, the workflow WF_2 depicted in Figure 2.1(b) is an unstructured workflow. Obviously, it is challenging to describe workflow WF_2 by using structured definition. Some research work [95, 160, 74, 133] investigate the transformation of an unstructured workflow to a BPEL-compatible structured workflow. However, it is challenging to prove that workflow transformation can succeed for all kinds of unstructured workflow Moreover, the transformation process can bring additional cost and complexity (from both design time and runtime).

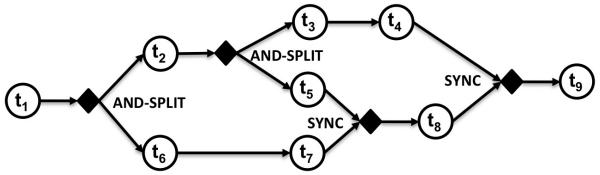
Additionally, the structured process definition is rigid and hard to maintain. As an example, due to the modification of business policy, the SBA provider wants to modify workflow WF_2 to WF_2' depicted in Figure 2.1(c). Using traditional approaches, such modification of workflow structure presents a high level of complexity, especially at runtime. Moreover, sometimes, a structured workflow can be modified to an unstructured one. In this case, a tiny modification on the structure of workflow can lead to the completely redesign of the entire process.

Finally, the execution of such a business process is based on centralized execution engines, which may raise the performance limitation in execution time and throughput. In response to these problems, some research work have proposed the decentralized execution of service compositions, as we have discussed in Section 1.3. However, the partitioning of workflow is still a complex and challenging task by using these traditional approaches. Moreover, it lacks generic partitioning approaches that can be applied to all kinds of workflow with little human's involvement. Finally, the distribution of workflow fragments is often performed at design time rather than runtime. Such statical decentralized execution model can hardly react to the ever-changing execution environment.

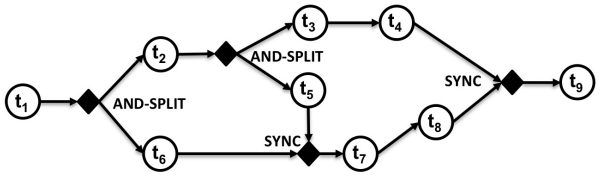
Thereby, new paradigms are required to meet the needs for flexible expression and execution of service composition. In this section, we introduce both rule-based systems and tuple space based systems as a promising paradigm for flexibly expressing service coordination.



(a) Workflow WF1



(b) Workflow WF2



(c) Workflow WF2'

```
<sequence>
    <task name=t1>
    <flow>
        <sequence>
            <task name=t2>
            <flow>
                <sequence>
                    <task name=t3>
                    ...
                    <task name=tm>
                </sequence>
                <sequence>
                    <task name=t4>
                    ...
                    <task name=tn>
                </sequence>
            </flow>
            <task name=tj>
        </sequence>
        <sequence>
            <task name=t5>
            ...
            <task name=tl>
        </sequence>
    </flow>
    <task name=tp>
</sequence>
```

(d) Structured Definition of WF_1

Figure 2.1: Structured Workflow Definition

2.1.1 Rule Based Systems

As stated in [53], "The most significant changes do not come from re-engineering workflow, but from rethinking rules". As a result, the rule-based system is considered as an appropriate approach for workflow management due to better flexibility and adaptability [115, 126, 125]. Using rule-based approach, the execution logic between any two services (expressed by the edges between any two tasks) are expressed by a rule. As an example, the definition of rule R_1 below describes the sequential execution order between two tasks S_6 and S_7 of the workflow WF_2 given in Figure 2.1(b).

$$R_1 = \textbf{if}\ \text{task}\ t_6\ \text{is executed}\ \textbf{then}\ \textit{execute}\ \text{task}\ t_7$$

Complex workflow patterns can also be expressed in the similar way. For example, the computation of task t_5 and t_7 have to be synchronized before executing task t_8, which is dictated by the rule R_2 defined below:

$$R_2 = \textbf{if}\ \text{task}\ t_5\ \text{is executed}\ \&\ \text{task}\ t_7\ \text{is executed}\ \textbf{then}\ \textit{execute}\ \text{task}\ t_8$$

By this means, a workflow is described by a number of rules rather that a hierarchy of structured tasks. Each rule only describes the relationship between a task and its precedent(s)/successor(s). Therefore, unstructured workflows can also be expressed.

Additionally, the workflow can be flexibly managed by manipulating these rules. Consider the example addressed before, the modification of the workflow WF_2 to WF_2' (refer to Figure 2.1) only needs the modifications of few rules. Since a synchronization pattern is inserted before task t_7, the rule R_1 is simply rewritten to R_1', defined as follows:

$$R_1' = \textbf{if}\ \text{task}\ t_6\ \text{is executed}\ \&\ \text{task}\ t_5\ \text{is executed}\ \textbf{then}\ \textit{execute}\ \text{task}\ t_7$$

Meanwhile, the rule R_2 is re-write to R_2' that expresses the sequential execution order between task t_7 and t_8:

$$R_2' = \textbf{if} \text{ task } t_7 \text{ is executed } \textbf{then } \textit{execute} \text{ task } t_8$$

From this example, we can see that the rule-based approaches exhibit a high degree of flexibility in both expressing and managing workflows. In this section, different rule-based systems are introduced.

2.1.1.1 Pure rule-based system.

[115] introduce a framework for business rule driven service composition. First, the authors introduce a phased approach for developing and managing service compositions. The life-cycle of service composition consists five phases, including definition, scheduling, construction, execution and evolution. Then, a framework is proposed to implement the entire life-cycle as rule-based systems. The construction and execution of a service composition is managed by a set of structure rules, data rules, constraint rules, resource rules and exception rules. Finally, the authors argue that rule-based approach is more flexible and dynamic compared to traditional approach and it can also reduce the time and effort to develop and manage service compositions.

A rule-based workflow management system for Web service composition is presented in [147]. The coordination of Web services is expressed by a set of Event-Condition-Action rules (ECA-rules). ECA-rules are originally used in active database systems [119]. An ECA-rule is triggered by a predefined event E, if the condition C holds, the action A is finally executed. The event can be either a primitive one or a composite one (by composing multiple primitive events). The authors argue that ECA-rules are suitable for expressing workflow because 1) it is easy to understand for the developers and 2) it can easily express complex workflow. Later in [55], this work is extended by introducing the formalization of ECA rule-based workflow modeling and an algorithm for event composition/decomposition. Finally, a real WFMS is implemented with a Graphic User Interface (GUI) that helps users to quickly and easily develop workflows.

[148] introduces a framework for adaptive orchestration (FARAO), which uses a set of Condition-Action (CA) rules to manage the data flow. CA rules are created by three steps. First, a number of CA rules can be automatically derived based on the WSDL files of all constituent services, which reflect the data dependencies between services. Then, these rules are extended by merging business rules to determine the decision making during the execution of workflow. Finally, users can also add additional control-flow constraints to express the execution orders, e.g., execution sequence. The authors state that one of the advantages is that, some CA rules are generic so that they can be reused for different workflows.

2.1.1.2 Hybrid rule-based system.

However, as discussed in [53, 125], the pure rule-based system is not appropriate since there is no global view on the service composition. Furthermore, it is separated from the standard approach, namely WS-BPEL. So that the users are always required to learn new knowledge on how to write rules to express business logic. As a result, the authors propose a hybrid approach for expressing Web service compositions by integrating the rule-based systems to the existing BPEL-based approaches.

People argue that it is a good way to separate the design of data flow and business rules. Since the ultimate business goal does not change frequently, the data flow is thus hardly modified. By contrast, business policies are determined by the global market, which evolves fast. Different policies achieve the ultimate business goal by selecting different execution

paths. In this context, the separation of data flow and business rules can greatly reduce the complexity to maintain service composition. SBA provider only need to modify the implementation of business rules (or add new rules) according to the execution context (market).

[53] presents a hybrid approach, which breaks the service composition logic into business process and business rules that exist and evolve independently. Two alternative implementations of business rules are introduced. One is based on Aspect-Oriented Programming technique that is able to weave business rules into BPEL frames [54] and the other is based on Business Rule System (BRS) [126, 105].

Another similar work [125] presents how to integrate business rules into BPEL specification in a service-oriented way. This paper indicates that it is hard to realize the direct communication between a BPEL engine and a business rule engine. So the authors propose to expose business rules as services, which increases the reusability of rules within an enterprise domain. An Enterprise Service Bus (EBS) [51] is proposed as a middleware for integration platform.

2.1.2 Tuple-Space Based Systems

People do exchanges quite often in the daily life. For example, in the colleges, students often borrow books or lecture notes from others; professors need distribute teaching materials to the students; boys often send cards to girls. However, face-to-face delivery each time is not efficient in our busy daily life. In this context, people do the exchange in a more effective way: using mailbox. Often, a college has a wall of mailboxes, where each student has his/her personal one. If you want to give something to others, you just need to put it in his/her mailbox. On the other side, you may also find the surprise in your own mailbox.

Tuple space, working as the mailboxes, is a repository of tuples that can be accessed concurrently. Thus, it is an asynchronous interaction paradigm for distributed and parallel computing. As an illustrative example, consider that there are a group of processors that produce pieces of data and a group of processors that use these data. Producers post their data as tuples in the space, and the consumers then retrieve data from the space that match a certain pattern[1]. Linda [10] is a coordination language that implements tuple-space based model.

Some research work investigate the use of such tuple-space based approach for service coordination, named as "tuple-space based Web Service composition". By using the tuple space, the interaction between services can be greatly decoupled. The general idea is that each service manages a tuple space. If a service S_R wants to call another service S_C, it just needs to write a request tuple into the tuple space of S_C. S_C is able to get the request tuple from its space and then do the related jobs to fulfill the expected functionality. After the computation, S_C will write back the results into the tuple space of S_R and so that S_R is able to get it. (It works exactly similar to the mailbox example.)

[70] presents an infrastructure and the corresponding tools named TSSuite to support web service modeling, design and management. TSSuite works based on TSpace [151], which extends the tuple space to handle web services. As a proof of concept, the authors present an intelligent printer system, where each service manages a tuple space. If there are some request tuples in the space, services will perform the real action or call another service. By this way, the workflow is divided into several sub workflows, which are distributed in the internal or external region of an organization. This has greatly

[1] From Wikipedia. `http://en.wikipedia.org/wiki/Tuple_space`

enhanced the fault-tolerant and parallel access. The tuple-space based orchestration can be improved by using notification. When a service has finished its job, it writes back the results into the tuple space as well as notifies the original business process to come and get it. By this mean, the service requester need not periodically check if it the result has come out.

[97] presents a tuple space for XML and its implementation in Web service orchestration, named xSpace. The authors give an example for business process orchestration. In this example, the workflow controller uses xSpace to put in requests for the next workflow task and reads the results from xSpace before deciding the next executing step. Different kinds of XML documents describe the results of different steps and the controller does an associative search to determine whether any document with the expected type is available for processing. Currently, the ongoing research is to integrate a notification system to xSpace.

2.2 Chemistry-Inspired Computing

Rule-based systems present a high flexibility in expressing and managing service compositions, and tuple space based systems promote the synchronization in the interactions between services. According, we began to think: how about combining both of them?

In this section, we are going to introduce another unconventional approach known as chemical computing [30, 29, 24, 28], which shares a lot of similarities with both rule-based and tuple space based approaches. It is designed originally for parallel and autonomic computing. Inspired from chemistry, computation is described as a series of chemical reactions. A program is described as a *chemical vessel* (or a *chemical solution*) that contains mixed chemical substances defined in terms of atoms/molecules. The chemical solution is implemented by the *multiset*[2], which extends the concept of *set* with multiplicity. An element can be presented only once in a set whereas many times in a multiset. The time of its occurrence is defined as *multiplicity*. For example, {1, 3, 1} is a multiset but not a set because the multiplicity of the integer "1" equals to 2.

All the molecules, defined by a number of program objects (e.g. Java objects), represent the computing resources, such as data. The computation (e.g. data processing) is performed by a series of chemical reactions controlled by a set of rules. Similar to chemical equations that reflect nature laws by specifying the reactant and resultant of a chemical reaction, a rule implements a specific algorithm by defining the input and output of a certain computation process. In this context, a chemical reaction is implemented as a multiset rewriting process [88], which can generate a (number of) new molecule(s) by consuming a (part of) existing one(s) in the solution. Chemical reactions are performed as the falls of dominos tiles: the output molecule(s) of a rule may activate other rules and trigger a series of new reactions in succession. The computation completes when the solution becomes *inert* - that is to say, no rule in the solution can still be active, and so that no reaction can be triggered any more. When such a stable state is reached, the final result is obtained and left in the multiset. The metaphor of chemical computing is summarized in Table 2.1.

Chemistry-inspired computing shares a great degree of similarities with both rule-based and tuple-space based approaches. On one hand, the core of a chemical program is the definition of rules that direct the computation. On the other hand, the reactions take place in chemical solutions, which performs like a tuple space. Accordingly, by taking

[2]In the remaining part of this dissertation, we use the terms *solution* and *multiset* interchangeably.

Table 2.1: Metaphor of Chemistry-Inspired Computing

Traditional Approach	**Chemical Metaphor**	**Chemical Implementation**
program	chemical vessel (solution)	multiset
computing resources	molecules	program objects
algorithms (control)	chemical equation	rules
computation	chemical reactions	multiset rewriting

the advantages from both approaches, we believe that chemistry-inspired computing can be a promising candidate for programming service-based system. In the following, our discussion starts with the the evolution of chemical programming models that have implemented chemical computing. And then some chemistry-inspired approaches for modeling autonomous and adaptive service-based systems are introduced.

2.2.1 Gamma

To the best of our knowledge, Gamma is the first chemical programming model proposed in 1986 [26] and later extended in [27] as a formalism for parallel computing through a chemical metaphor. The computation in Gamma is performed by a series of multiset rewriting processes controlled by a set of rules. A rule defines a chemical reaction by specifying how to rewrite the multiset, namely the consumption of the existing molecules and the creation of new ones. The definition of a rule is composed of 2 parts: *condition* and *action*. If the *condition* is held by a portion of elements, they will be replaced by other elements according to the *action* part. As a simple example, to compute the maximum number in a non-empty multiset, the following rule is defined:

replace x, y **by** x **if** x>= y

Under the control of this rule, any two integers, say x and y respectively, will react and the smaller one will be removed (in case that two integers have the same value, any one of them can be removed). Then if this rule is put into the following multiset: {1, 2, 4, 6, 5, 7, 8, 9, 9}. x and y can match any pair of integers, for example, $x \leftarrow 4$, $y \leftarrow 1$. In this case, the integer "1" will be removed from the multiset. Accordingly, when the computation completes, only the maximum number (9) can be finally left in the multiset.

Gamma expresses implicitly the parallelism. The programmers do not need to explicitly specify the sequentiality of reactions. At runtime, a rule can autonomously direct multiple reactions at the same time. Hence, different reactions can take place simultaneously and independently. Under this context, the execution of a Gamma program is non-deterministic. The molecules react randomly with others. Therefore, different executions of a Gamma program may result in different execution orders (sequences of reactions). More introduction about Gamma formalism can be found in [28].

2.2.2 γ-Calculus

γ-calculus [29, 24] can be seen as a higher order extension for Gamma. The syntax of molecules in γ-calculus is shown in Figure 2.2. A molecule can be any of the following four types: 1) a variable x; 2) a γ-abstraction which stands for a reaction rule by replacing a molecule of pattern P by molecule M; the pattern P can match either a molecule, a compound molecule or an inert solution; 3) a compound molecule built with the associative

```
M :=  x                  ; variable
   | (γ(P).M )           ; γ-abstraction (reaction rule)
   | ( M1, M2 )          ; compound molecule
   | ( <M> )             ; solution

P :=  x                  ; matches any molecule
   | P1,P2               ; matches a compound molecule
   | <P>                 ; matches an inert solution
```

Figure 2.2: Syntax of Molecules in γ-Calculus

```
γ-Reduction:
     (γ<x>.M),<N> →γ M[x:=N] if Inert(N) ⋁ Hidden(x,M);
α-Conversion:
     γ<x>.M ≡ γ<y>.M[x:=y] with y fresh;
Commutativity:
     M1,M2 ≡ M2,M1 ;
Associativity:
     M1, (M2,M3) ≡ (M1,M2),M3 ;
```

Figure 2.3: Rules of γ-Calculus

and commutative constructor (,); 4) a solution which isolate the molecule M from other molecules.

Figure 2.3 lists some rules of γ-calculus. The γ-reduction rule applies the γ-abstraction (reaction rule) to a solution, which can extract its inside molecules. When the rule γ<x>.M is applied to a solution <N>, either of the following two conditions has to be satisfied: 1) *Inert (N)*: the content N of the solution argument is a closed term made exclusively of abstractions or exclusively of solutions (which may be active). 2) *Hidden (x, N)*: the variable x occurs in M only as <x>. Therefore <N> can be active since no access is done to its contents. While an α-conversion is like a method of substitution, we replace all x by y in M and build another form.

In the following, we provide a concrete example to illustrate how to apply these rules. First of all, we define a γ-abstraction *add_one* as follows, which add any integer by one.

$$add\_one = \gamma< x >.< x + 1 >$$

Then, we consider the following multiset which contains an *add_one* γ-abstraction and a solution with an integer 5, denoted as $add\_one, < 5 >$. Since the condition *Inert*$(< 5 >)$ is held, the γ-reduction rule can be applied. x maps to the integer 5, and after the reaction, the multiset contains only one element: $< 6 >$.

γ-calculus is expressive. A reaction rule can be defined as *one-shot* or *N-shot*. A one-shot rule can be applied only once, after the reaction, it is consumed. By contrast, a N-shot rule will never be consumed and can be applied by any number of times. The reaction rule *add_one* that we have defined in the previous example is a one-shot rule. As an example, an N-shot rule is defined to performed addition operation. The rule *add* is defined as follows:

$$add = \gamma< x >.\gamma< y >.< x + y >, add$$

Then, consider the following multiset: $add, < 2 >, < 3 >$. First of all, the molecule $< 2 >$ can react with the reaction rule add, after the first step of reaction, the multiset looks like:

$$< 3 >, \gamma < y >.< 2 + y >, add$$

Then, the new γ-abstraction can be applied on the molecule $< 3 >$. After the reaction, the content of multiset becomes:

$$< 5 >, \text{add}$$

We can see that after the addition operation, the rule *add* is put into the multiset again. Therefore, the reaction can be continued by one step more. Finally, the multiset contains only one γ-abstraction as follows:

$$\gamma < y >.< 5 + y >, add$$

Two fundamental extensions. γ-calculus is very expressive, whereas compared to the original Gamma and other chemical models, it lacks two fundamental features: 1) it can not conditionally trigger a reaction; 2) it can only react with one element at a time, if you want to react with several elements (i.e.: to perform an addition), you need to take several steps to get all the parameters and make the computations (such as the above example). In the following, we introduce two extensions to γ-calculus in response to these problems.

1. **Conditional reaction.** We rewrite the γ-abstraction as γ_c-abstraction by adding conditions to the abstractions. γ_c-abstraction is expressed as:

   $$\gamma\text{<x>}\lfloor M_0 \rfloor.M_1$$

   It is equal to the Gamma expression "replace x by M_1 if M_0". As a consequence, besides holding either of the Inert() or Hidden() condition, the reaction condition M_0 has to be tested as *true* before the reaction can start. Improvement with conditional reaction helps us to perform type checking and pattern-matching, which confines the reaction within a certain type of elements (e.g. operations only for integers). As an example, suppose that you want to decrease any positive integer by 1, the following γ_c-abstraction can be defined:

   $$decreaseByOne = \gamma < x > \lfloor x > 0 \rfloor.< x - 1 >, decreaseByOne$$

   The following example describes how γ_c-abstraction can be applied.

   $$decreaseByOne, < 0 >,< 1 >,< -1 > \rightarrow decreaseByOne, < 0 >,< 0 >,< -1 >$$

   With the requirement that the value of integer has to be greater than 0, the rule *contertToOne* can only react to $< 1 >$. After the reaction, $< 1 >$ is replace by $< 0 >$ and no reaction can be triggered any more.

2. **Atomic capture.** γ-abstraction can be extended to γ_n-abstraction so that it can capture multiple molecule as reactant, expressed as follows:

   $$\gamma \text{ (<x1>...<xn>).M}$$

   It equals to the Gamma expression: **replace** $x_1, ..., x_n$ **by** M. In this scenario, the reaction can occur only when it gets all the expected molecules; otherwise, no reaction will take place. As an example, using γ_n-abstraction, the addition operation can be described as follows:

$$add_A{=}\gamma(< x >, < y >).< x + y >, add_A$$

Compare the the formal example, the reaction only takes once, as illustrated below:

$$add_A, < 2 >, < 3 > \rightarrow_\gamma add_A, < 5 >$$

After the first reaction, since the γ_n-abstraction add_A cannot get two inert solutions with an integer, so the multiset becomes inert.

2.2.3 Higher-Order Chemical Language (HOCL).

In the following, we introduce a chemical programing language, named Higher-Order Chemical Language (HOCL) [23], which implements γ-Calculus.

2.2.3.1 HOCL Syntax

The syntax of HOCL is defined in Figure 2.4, which extends the previous models with types, pairs, empty solutions, naming and expressive patterns.

Types. Since a multiset may contain the molecules of different types (such as strings, integers, etc.), the extension of types is useful in pattern matching for the selection of the reactable molecules. The pattern-matching rule is expressed as:

$$\text{match(x::T,N)} = \{ \text{ x} \rightarrow \text{N}\} \text{ if Type(N)} \preceq \text{T}$$

As an example, the following reaction rule γ(x::int, y::int).x+y can be only applied to integers. In HOCL, any a type is a subtype of the *universal type*, denoted by $\star$ ($\forall$T, T$\preceq \star$). A variable of the universal type can match any numbers of molecules of any types, even for an empty one (the example of an empty molecule is provided later on).

Pairs. A pair is denoted by "P1:P2", a colon is used as chemical bond that connects two isolated molecules. The pattern-matching rule for pairs is:

$$\text{match((P1:P2),(N1:N2))} = \phi_1 \oplus \phi_2 \text{ if match(P1,N1)}{=}\phi_1 \bigwedge \text{ match(P1,N1)}{=}\phi_2$$

Generally, the notion of pair can be extended by using several colons to connect multiple fields. As a result, a pair can be defined in the following form (P1:P2:...:Pn). In this case, a pair is also called a *tuple*. A tuple can be regarded as a struct in C language, which, like an union, groups multiple variables into a single record.

Empty solution. The notion of empty solution in HOCL is raised since that after reactions, the multi-set might become empty. This is caused by the universal pattern. Considering the following example:

$$< 2, 3, 4, 5, 6 >, \gamma < x, \omega > .\lfloor x > 1 \rfloor .< \omega >$$

The reaction rule removes a number x if x is greater than 1. When there is only one integer left in the multiset, say 6, x will match 6 and ω will match an empty molecule, which means nothing. Therefore, by applying this rule to the multiset, all the numbers will be removed. After reaction, the solution becomes an empty one.

Solution

```
S ::= <M>                                   ; solution
   | <>                                     ; empty solution
```

Molecules

```
M ::= x                                     ; variable
   | M1, M2                                 ; compound molecule
   | A                                      ; atom
```

Atoms

```
A ::= x                                     ; variable
   | [name=]γ(P)⌊V⌋.M                       ; reaction rule, possibly named
   | S                                      ; solution
   | V                                      ; basic value
   | (A1:A2)                                ; pair
```

Basic Values

```
V ::= x | 0 | 1 | ... | V1+V2 | -V1| ...    ; integer
   | true | false | V1⋀ | ...              ; boolean
   | V1=V2 | V1≤V2 | ...
   | ‘‘String" | ...                        ; String or expressions
```

Patterns

```
P ::= x::T                                  ; matches an molecule of type T
   | ω                                      ; matches any molecule even empty
   | name = x                               ; matches a named reaction
   | <P>                                    ; matches an inert solution
   | (P1:P2)                                ; matches a pair
   | P1,P2                                  ; matches a compound molecule
```

Types

```
T ::= B                                     ; basic type
   | T1× T2                                 ; product type
   | ⋆                                      ; universal type
```

Basic Types

```
B ::= Int | Bool | String
```

Figure 2.4: Syntax of HOCL

Name. To facilitate its reuse, a rule can be named (or tagged) by using the syntax *name* = γ(P).$\lfloor V \rfloor$.M. The pattern-matching rule for named reactions rule is:

$$\text{match}((\text{name=x}),(\text{name=N})) = \{ \text{x} \rightarrow \text{N}\}$$

More details about the introduction to HOCL syntax can be found in [23].

2.2.3.2 HOCL Grammars

In this part, we describe how to write HOCL programs. An HOCL program is composed of two parts: *rule definition* and *solution organization*.

Rule definition. The definition of a chemical rule follows the pattern:

$$\textbf{let } \textit{rule\_name} = \textbf{replace(-one)} \ P \ \textbf{by} \ M \ \textbf{if} \ C$$

- The **let** keyword assigns a name to a definition of chemical rule.
- A rule can be either *one-shot* or *N-shot*. Using HOCL, *N-shot* rules are defined with the **replace** keyword whereas *one-shot* rules are declared by using the **replace-one** keyword.
- Next, the **replace(-one)** keyword defines the patterns of a list of input molecules P (as the reactants). Each input molecule is defined with a specific type using double colons (noted as *varName*::*type*). Both primitive types (e.g. integer, string, etc) or complex types (e.g. Java objects) are supported. During the execution of a chemical reaction, each variable is mapped to a molecule of the required type.
- On the other side, the **by** keyword indicates how to generate the output molecules (as the resultant). The existing molecules can be modified here, or even new ones can be generated.
- Finally, the **if** keyword is optional which denotes the condition under which the reaction can take place.

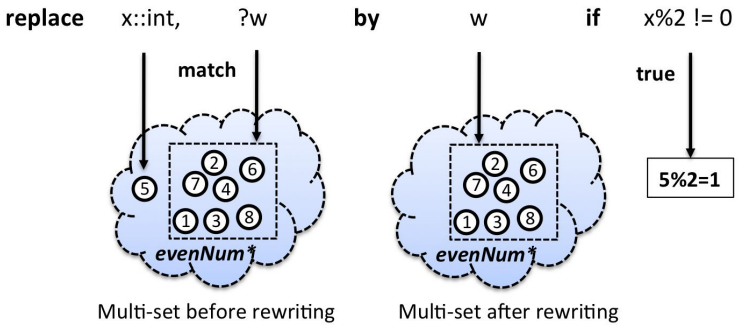
Solution organization. The chemical solution is implemented by a *multiset*, expressed by a pair of marks "<" and ">"; in between all the elements are defined as molecules, such as data, rules as well as sub-multisets. A molecule can be either *atomic* or *complex*. An atomic molecule represents a basic individual unit that takes part in the reactions, such as an integer. Particularly, chemical rules and inert sub-solutions are two kinds of special atomic molecules, which can also be manipulated by other rules (explained later on). By contrast, a complex molecule is a collection of atomic molecules, denoted as a *tuple*: M_1:M_2:...:M_n. A single colon is used as the chemical bond that connects two atomic molecules. More information about how to write HOCL programs can be found in [144].

2.2.3.3 Sample HOCL Program

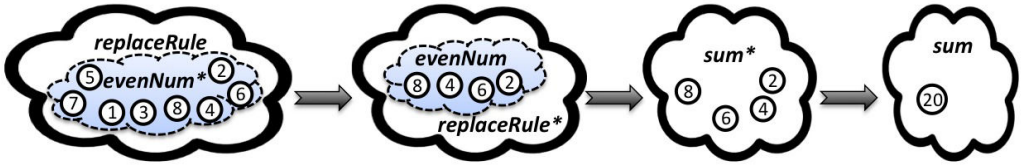
In Program 2.1, a sample HOCL program is defined to compute the sum of all the even numbers for a given set of integers. Three rules are defined: 1) firstly, the rule *evenNum* removes all odd numbers: it reacts with any an integer x, if x is not the multiple of 2, it will be removed. The variable w is defined as a *universal type* (using "?" mark), which represents all the remaining molecules in a (sub-)solution. As illustrated in Figure 2.5(a), w represents all the other integers except x in this case. 2) Once all odd numbers have been removed, the sub-solution becomes inert. Thus it can be manipulated as a normal atomic molecule. The rule *replaceRule* will extract all the even numbers (represented by w here) from it, and replace the rule *evenNum* by a new rule *sum*. Please note that *replaceRule* is a *one-shot* rule (defined by the keyword **replace-one**), which can be applied only once since it is consumed after the reaction. 3) In the following, the rule *sum* performs the addition operation by replacing any pair of integers with their sum. The execution sequence of this sample program is vividly illustrated in Figure 2.5(b), the active rule for each step is marked with a star (*). After all reactions have been completed, the global solution reaches the inert state and the final result is left in the solution.

Program 2.1: A Sample HOCL Program

```
let evenNum =
  replace x::int, ?w
  by w
  if x%2!=0
let sum =
  replace x::int, y::int
  by  x+y
let replaceRule =
  replace-one <evenNum, ?w>
  by  sum, w
<replaceRule, <evenNum, 1, 2, 3, 4, 5, 6, 7, 8>>
```



(a) The Execution of the Chemical Rule *evenNum*



(b) The Execution Sequence

Figure 2.5: The Illustration of the Sample HOCL Program

2.2.4 Chemistry-Inspired Service Systems

From the above-introduced example, we can conclude that the execution of a chemical program has the following characteristics:

1. **Implicitly parallel**: a number of reactions can take place independently and concurrently to each other.
2. **Non-deterministic**: for each reaction, the reactant molecules are chosen randomly;
3. **Higher-order**: the execution is able to react to runtime execution context by replacing old reaction rules by new ones, which present new computational algorithms/policies.
4. **Evolving**: new molecules, either rules or data, can be put into a chemical solution at runtime and then immediately take part in the chemical reactions.

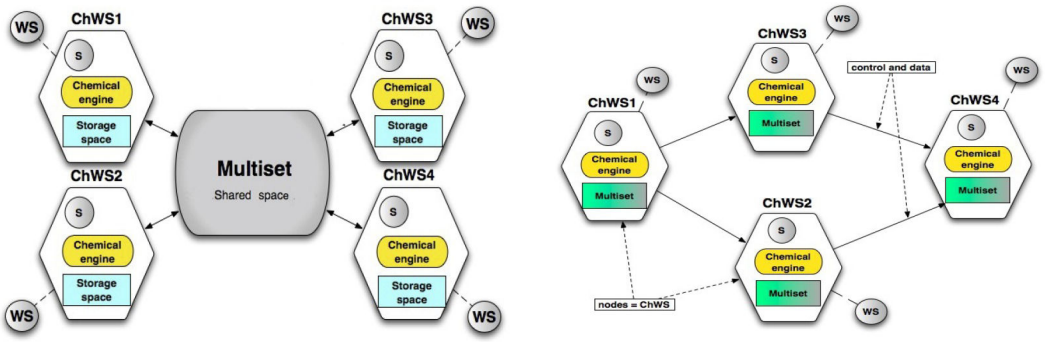
All these characteristics share a large degree of similarities with the objective of building self-adaptive and long-lasting evolving service-based systems, for example, flexible management of SBAs. Some preliminary work have investigated and discussed the viability of using chemical programming model to express service coordination. [108] presents a highly abstract coordination framework for distributed workflow enactment based on γ-calculus. The coordination is performed in single global multiset named coordination solution, inside of which, resources are presented by the sub-solutions. It is demonstrated that all basic and complex workflow patterns can be expressed in a declarative way using the γ-calculus. The authors extend this work in [109] by introducing fault tolerance, resource control, constrained resource matching, etc.

In addition, some research work investigate the use of HOCL for modeling service coordination. [25] demonstrate how to use HOCL rules to express the coordination of resources for executing desktop grid-based applications. Through a simple example, the authors argue that HOCL can be a good candidate for grid programming and coordination due to some desirable properties such as implicit parallel and autonomic nature. Later, [32, 31] investigates the utilization of HOCL to express service coordination by developing a simple but practical example. This work demonstrates that complex coordination structures such as sequence, parallel, mutual exclusion, etc, can be expressed using HOCL rules. Based on this work, in [139], a framework is proposed to define the workflow and to express service orchestration using chemical concepts. The authors have proven that most of WS-BPEL constructs, such as invoke, reply, receive, sequence, flow, thrown etc., can be expressed using HOCL rules.

Additionally, other research work aims to use chemical programming to construct an executable concrete workflow by means of selecting appropriate services [71, 59, 62, 60]. A task can be executed by multiple functional-equivalent services. And each service e^i can be delivered by a time interval Δt_{e^i}=T_{e^i}-t_e^i. The service selection aims at selecting a service for each task so that the execution of workflow can successfully complete. The authors modeled workflow as a complex molecule and each service candidate acts as an atom. The service selection is performed by a series of chemical reactions that connect each task molecule to a suitable service atom, with the consideration of time constraints and dependencies of any two adjacent tasks. The selection is an evolving process that new offers can come freely at runtime. Later in [60], the authors extend this work by introducing lazy instantiation of workflows. The workflow can be partially instantiated due to some conditional constructs, such as if-then nodes. However, these work aims at seeking a feasible solution rather than a (sub-)optimal one.

The work presented in [67, 68, 69] share some similarities with our work. First, the authors point out that most of afore-mentioned work relies on a centralized multiset as the coordination space. And they argue that this centralized architecture suffers from a poor scalability, low reliability, communication bottlenecks and privacy issues. In response to this problem, two decentralized chemical frameworks are proposed for the coordination and the execution of service composition. 1) Figure 2.6(a) illustrates the HOCL-TS WMS architecture, a shared-space multiset is used as the communication mechanism for sharing coordination information and data among all Web services. This work is validated later in [69] by implementing a scientific workflow management system. However, the shared-space multiset is still implemented by a centralized multiset, which is publicly writable and readable. This centralized implementation suffers from poor scalability. 2) The HOCL-P2P WMS architecture is shown in Figure 2.6(b), each constituent service can directly talk with each other. This work is similar to both choreography models proposed in this dissertation

[3]Both figures are borrowed from [67].



(a) HOCL-TS WMS architecture

(b) HOCL-P2P WMS architecture

Figure 2.6: Decentralized Chemical Frameworks for Service Coordination [3]

(introduced later in Section 3.4). However, HOCL-P2P WMS architecture requires the link between any two constituent services to be predefined at design time. Therefore, it is incapable to meet the requirement for building flexible service-based system, where constituent services are selected at runtime. Additionally, no failure recovery nor adaptation issue has been addressed. Our models exhibit better flexibility: the configuration, coordination or even adaptation are performed at runtime, in an autonomous way. This work can be integrated with the distributed chemical runtime [113, 110, 111, 112], By building an additional layer based on the peer-to-peer communication protocol [129], the execution of HOCL programs can be easily deployed over a large-scale infrastructures.

2.3 Illustrative Example: The "Best Garage"

In this dissertation, we use an automobile repair shop as a living example to illustrate our approach: the *"Best Garage"*[4] provides car reparation services for its customers. An SBA is implemented for autonomous management of the reparation process. The reparation of a customer's car follows the workflow illustrated in Figure 2.7, which coordinates a collection of interrelated *task*s:

- First of all, an evaluation is performed in task t_1 to preliminarily determine the problem of the car (noted as $Prob_{car}$) and to estimate the complexity of the reparation (in terms of reparation time and cost).
- In the following, the reparation is continued to one of the two exclusive branches. Based on the preliminary diagnostic, the garage evaluates whether it is capable to deal with the problem. If it could, the car is then sent to a specific workshop for the reparation (task t_2).
- Otherwise, the car will be sent to a partner garage to outsource the reparation of the car. The price and expected time consumption for the reparation are also estimated. However, the outsourcing process is not aware by the customer.
- Finally, the customer is required to pay the cost of bill in task t_4 after the completion of reparation.

[4]The "Best Garage" is a fictitious automobile shop which does not really exist. Any existences of actual automobile shops with the same name are coincidental.

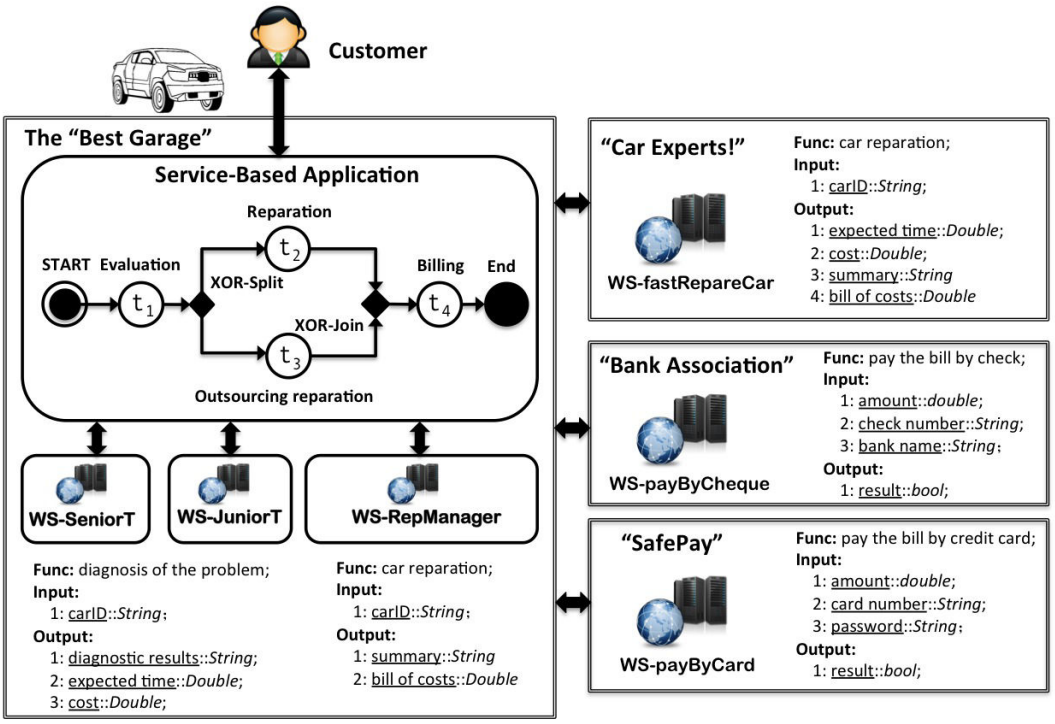


Figure 2.7: Illustrative Example: the "Best Garage"

Each task can be executed by a Web service. As shown in Figure 2.7, a number of functional-equivalent services co-exist for executing a task.

- First, either a senior technician or an assistant technician can be invited for the evaluation. The "Best Garage" has developed two Web services, *WS-SeniorT* and *WS-JuniorT*, to schedule respectively senior and assistant technicians. When a request arrives, the Web service sends a message to an available technician with the job information and marks his/her status as "occupied". When a technician finishes the evaluation job, his/her status is changed to "available" again so that (s)he is able to accept new jobs.
- The reparation of a car in the "Best Garage" is managed by Web service *WS-RepManager*. It requires the identity of the car as input, and reserves the available resources for the reparation, such as workshop, engineers etc. After the reparation, it returns the summary of reparation and the bill of costs.
- In case that the reparation is not able to be solved locally, the "Best Garage" seeks for the solution from its partner automobile repair shop named "*Car Experts!*"[5]. The partner garage provides a Web service *WS-fastRepareCar* which accepts a request for reparation as input, then analyzes the problem and proposes the price and the expected time. After the reparation, the car will be returned to the "Best Garage" with a summary of reparation and the bill of costs.
- Finally, multiple possibilities are provided to the customer for the payment. For example, the *WS-payByCard* Web service enables the customer to pay by credit card; whereas the *WS-payByCheque* allows the customer to pay by check.

[5]Similar to the "Best Garage", any existences of actual automobile shops with the same name are coincidental.

This example uses a simple workflow to illustrate service collaboration across multiple administrative domains. First of all, for each task, there may be multiple functional-equivalent services available (e.g., task t_1 and t_4). Then, different services may provided either internally by the "Best Garage" (e.g., *CWS-SeniorT*), or externally by different third-party service providers (e.g., *CWS-payByCard*). Moreover, some Web services are also implemented as composite web services that assembles and coordinates a number of services to create a value-added service. For example, *WS-RepManager* may coordinates several sub-tasks in order to manage the entire reparation process. However, from the requester's point of view (i.e. the "Best Garage"), it acts as a normal atomic service. Finally, we can see that everything can be a service. Behind the interface of a Web service, the concept of service means a wide range of things. For example, human resources (technician), real generic service (e.g. reparation service) or computational functionality (e.g., online payment/money transfer services).

Part II

Chemistry-Inspired Middleware

Chapter 3

Chemistry-Inspired Middleware for Flexible Execution of SBA

Abstract. In this chapter, we introduce a middleware inspired from chemical computing for flexible execution of SBA. The service-based system is described and implemented using the chemistry-inspired metaphor. First of all, the architectural overview of the middleware is introduced in Section 3.1. Using the chemical metaphor, the middleware can be seen as a distributed and autonomic chemical system: service selection, interaction, coordination and adaptation are modeled as a series of pervasive chemical reactions performed by molecular polymerization, decomposition and movement. Later in Section 3.2, we address the context-aware service selection problem. In order to construct an executable concrete workflow (represented by a service composition), the selection of suitable services can be perform at runtime with the consideration of both local and global constraints. Having a concrete workflow constructed, Section 3.3 and Section 3.4 present respectively centralized and decentralized models for executing and adapting service compositions.

3.1 Architecture of Chemistry-Inspired Middleware

The architectural overview of the chemistry-inspired middleware is illustrated in Figure 3.1. The middleware represents an additional layer between the (service-based) application layer and the (service) implementation layer. It can be seen as a medium by which Web services can be coordinated and consumed by service-based applications. Since the middleware shares the responsibility for runtime management of SBA (e.g., selection, coordination and adaptation of services), the application layer becomes thinner and the burdens of SBA providers can be reduced.

Using the chemical metaphor, the middleware is modeled as as a global service pool where a number of chemical abstractions of services are floated, named as *chemical service*s. Each chemical service represents a self-contained software agent. Accordingly, a chemical service is defined by a chemical solution of molecules that acts as a self-managed autonomic chemical system. Similar to a real-world Web services, a chemical service can also be either *atomic* or *composite*, according to the different roles that it plays.

- *Chemical Web Service (CWS).* A CWS is the middleware-level image of a real-world Web service. It refers to an atomic chemical service, which is the basic and indivisible functional unit in the middleware. A CWS is able to provide the expected functionality to other chemical services in the middleware by outsourcing the invocations to a connected real-world Web service. Therefore, it acts as a service consumer for a real-world Web service as well as a service provider in the middleware.

- *Chemical Composite Service (CCS).* By contrast, a CCS represents middleware-level service composition, which realizes the expected functionality by assembling and coordinating a number of chemical services in the middleware. As a result, it plays the role of service consumer in the middleware whereas service provider for the SBA application layer (the end requesters of SBA).

An SBA is mapped to a CCS in the middleware, which implements a solution of molecules to express the workflow and the coordination of chemical services. Service coordination is performed by means of interactions among chemical services in the middleware. The middleware-level service interaction is modeled as the movement of molecules from one chemical solution to another (as depicted by the dotted arrow in Figure 3.1). In this section, we present how to express (chemical) services and their interactions using chemical concepts.

3.1.1 Chemical Web Service (CWS)

Figure 3.1 illustrates four CWSes for the constituent Web services in the "Best Garage" example. A CWS can be seen as the chemical reflection of a real-world Web service. A one-to-one mapping is established between a CWS and the corresponding real-world Web service. A CWS acts as a bridge by which a real-world Web service can be consumed in the middleware. On one side, it receives invocation requests from other chemical services and then forwards the invocation messages to the connected Web service; one the other side, it receives the computational results from the connected Web service and then returns them to the corresponding requesting chemical service(s).

A CWS can connect to either an atomic Web service or a composite one. For example, *WS-SeniorT* Web service in Figure 3.1 may be developed originally as a software module that schedules the available senior technicians. And then this software module is made available on line as a Web service. By contrast, *WS-fastRepairCar* Web service can be

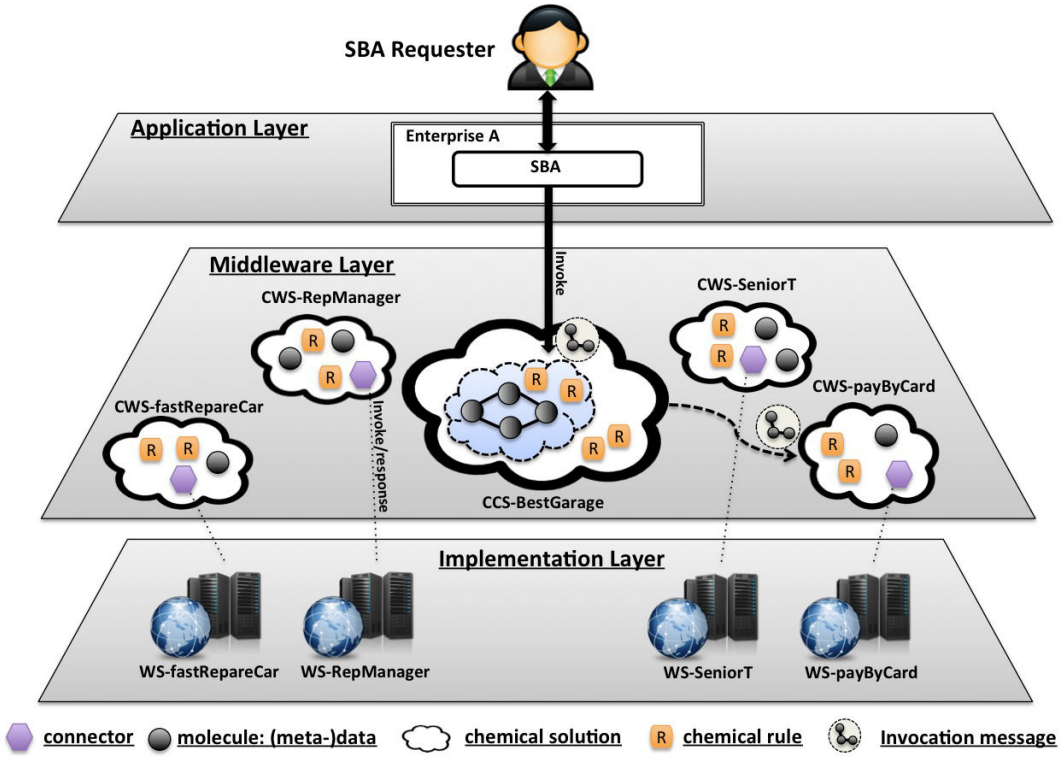


Figure 3.1: Architectural Overview of the Chemistry-Inspired Middleware

implemented as a service composition, defined by WS-BPEL and executed by one of the BPEL engines introduced in Section 1.3.1. However, the detailed implementation of workflow is not visible to the corresponding chemical service *CWS-fastRepareCar*. As a result, from the perspective of middleware, all real-world Web services (both atomic and composite ones) are regarded as atomic services .

A chemical Web service is implemented by a chemical solution where all floated molecules represent the meta-data (e.g. endpoint reference and interface descriptions) of the connected Web service. Each CWS solution contains a special molecule named *connector*, which takes charge of interacting with the connected Web service. The connector is a Java object which implements a Java Web service client for the connected Web service. By manipulating the connector, a number of rules are defined to control the interactions with both real-world and chemical services. Once a CWS receives an invocation request from another chemical service in the middleware, these rules are able to pick up the invocation parameters from its solution and then pass them to the connector. The connector translates the molecule-based invocation message to the standard one, and then forwards it to the connected Web service. Later, when the result from the connected Web service arrives, it will be translated again to a set of chemical molecules and finally written back to the solution of the corresponding chemical service. Concrete examples will be introduced in Section 3.3.

3.1.2 Chemical Composite Service (CCS)

Instead of interacting with a real-world Web service to provide the expected functionality, a Chemical Composite Service (CCS) reuses and assembles the functionalities provided by other chemical services to achieve the ultimate functionality required by the corresponding

SBA. In this context, an SBA is modeled as an autonomic chemical system that expresses a service composition using chemistry-inspired concepts: the workflow topology is abstractly modeled as a compound molecule and service coordination is expressed by a set of rules.

Molecular representation of workflow. The workflow specifies data flow by coordinating a collection of interrelated tasks. Using chemical metaphor, each task is defined by a task complex molecule, denoted by a *task tuple*:

"Task":*task_id*:*func*:*stat*:<"In":wfp_in:<>,"Out":wfp_out:<>>:*cws_sol*

A task tuple is composed of six parts: 1) a constant string ("Task") acts as the declaration of a task tuple. 2) *task_id* (string type) is the unique identifier of a task in the workflow. 3) *func* (string type) indicates the functional requirement of this task. 4) *stat* is defined as an enumerated type with a number of enumerators indicating the runtime execution state of a task, such as "Abstract", "Concrete", "Executing", "Executed" etc. Later in this section, we will explain the meaning of each state. 5) The *"In"* and *"Out"* sub-solutions include respectively all the direct precedents and successors of this task. *wfp_in* and *wfp_out* dictate respectively the coordination relationship with all its precedents and all its successors (e.g. all successors have to be executed exclusively or concurrently). 6) Finally, *cws_sol* is the *signature* of the relative CWS that is bound to execute this task. The signature is the unique identifier of a chemical solution in the middleware, expressed as: *solName@IP_addr/host_name*. It can be used to identify and to locate a chemical solution in the middleware so as to pass a set of molecules, as explained later in Chapter 4.2. For the reason of simplicity, we use only *solName* in the rest of our discussion[1]. The binding reference *cws_sol* of an abstract task equals to *Null*.

Thereby, the workflow can be expressed as a *cell* that includes a number of task molecules, defined by a *workflow tuple*:

"Workflow":<>

A workflow tuple starts with a constant string ("Workflow") and followed by a sub-solution that contains a number of task tuples. As a concrete example, Program 3.1 provides the description for the molecular representation of the workflow used in the *"Best Garage"* example. Task t_1 and t_4 are abstractly defined and suitable services will be selected and integrated at runtime based on the specific execution context, as addressed later in Section 3.2. By contrast, the binding reference of task t_2 and t_3 are concretely predefined since each task can be executed by only one specific chemical web service, *CWS-RepManager* and *CWS-fastRepairCar* respectively. After the execution of t_1, either t_2 or t_3 can be executed. Therefore, the output workflow pattern of t_1 is defined by an XOR-split pattern. Similarly, the execution of t_4 can start whenever either t_2 or t_3 completes, the input workflow pattern of t_4 is expressed by an XOR-join (also know as *simple-merge*) workflow pattern [8]. Please note that we are able to describe both abstract workflow as well as concrete workflow using molecular representation.

Reaction rules for workflow execution. The execution of workflow is modeled as a series of chemical reactions controlled by a set of reaction rules. The CCS solution also defines a set of rules that describe service coordination, invocation and adaptation as a series of molecular decomposition, polymerization and movement processes. Based on the different functionalities and objectives, all the reaction rules can be classified into three categories:

[1] And we assume that any two chemical solution have different names.

Program 3.1: Molecular Representation of the Workflow in the *"Best Garage"* Example

```
"Workflow":<
  "Task":"t1":"diagnostic":"Abstract":<
          "In":"Start":<Null>,"Out":"XOR-split":<"t2","t3">>:Null,
  "Task":"t2":"reparation":"Concrete":<
          "In":"Seq":<"t1">,"Out":"Seq":<"t4">>:"CWS-RepManager",
  "Task":"t3":"reparation":"Concrete":<
          "In":"Seq":<"t1">,"Out":"Seq":<"t4">>:"CWS-fastRepairCar",
  "Task":"t4":"billing":"Abstract":<
          "In":"XOR-join":<"t2","t3">,"Out":"END":<Null>>::Null,
  CRs, IRs, ARS
>
```

- **Coordination Rules (CRs).** The coordination rules schedule the execution of workflow tasks by deciding when to start the execution of a task based on the execution states of the other task(s). In Section 3.3.2, a number of CRs are presented to express complex workflow patterns.
- **Invocation Rules (IRs).** Invocation rules manage middleware-level service invocations. Each task t_i is associated with an invocation rule, noted as *invokeTi*. An invocation rule takes charge of preparing and sending an invocation message to the corresponding chemical service. More examples of IRs can be found in Section 3.3.
- **Adaptation Rules (ARs).** Adaptation rules react to runtime failures during the execution of workflow. Failures can come from either functional level (e.g. a responseless constituent service) or non-functional level (e.g. a late response). SBA provider can express various adaptation plans using ARs for different execution contexts. Runtime adaptation issue will be addressed in Section 3.3.3.

Creation of a new SBA instance. As shown in Figure 3.1, a request to execute the SBA will result in the creation of a new SBA instance that serves the corresponding SBA requester. Once a new SBA instance is created, the SBA writes an *invocation demand* to the solution of the corresponding CCS with 1) a unique identification string for this SBA instance and 2) the initial parameters provided by the end requester. An invocation demand is expressed as the following tuple:

"Invoke":*inst_id*:<parameter>

It consists of three parts: 1) a constant string ("Invoke") as the declaration of an invoke demand; 2) *inst_id* is the identification string of the relative SBA instance; 3) a subsolution includes all the invocation parameters provided by the end customers. Using the "Best Garage" as an example, a sample invocation tuple is provided as follows:

"Invoke":"CCS-BestGarage001":<"Car":<"Type":"RARE","ID":"ABCDEF">>

The emergence of such an invocation tuple in the solution of CCS will activate the rule *createSBAInstance*, as defined in Program 3.2. The creation of a new SBA instance is similar to the cell replication process, as shown in Figure **??**. A new instance tuple is created based on the predefined workflow tuple: all the contents of the workflow tuple, including the molecular workflow representation as well as all the rules for service

Program 3.2: The Definition of the Reaction Rule *createSBAInstance*

```
let createSBAInstance =
  replace "Invoke":inst_id::String:<?parameters>,"Workflow":<?wf_def>
  by  "Workflow":<wf_def>,
      "Instance":<"Instance_id":inst_id,wf_def,"Data":<parameter>, InstRs>
```

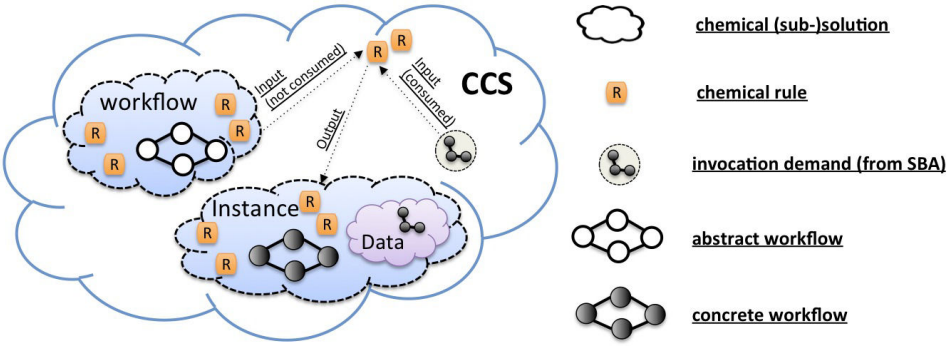


Figure 3.2: Illustration: Creation of a New SBA Instance

coordination, invocation and adaptation, are copied to the new instance sub-solution. Additionally, the new instance sub-solution contains some supplementary information. 1) A unique instance *inst_id*. 2) A "Data" sub-solution to stock the instance data, such as the initial parameters and the intermediate results of each task. 3) A number of instantiation rules (*InstRs*) that dictate how to select chemical services in order to construct a concrete and executable workflow (introduced later in Section 3.2). As a proof-of-concept example, in response to the above-mentioned invocation tuple, Program 3.3 provides the description of a new SBA instance created for responding the above sample invocation demand.

Please note that the original workflow tuple defined in Program 3.1 is inert, but the sub-solution of an instance tuple is active due to the introduction of instantiation rules. In this context, once a new instance tuple is generated, a series of chemical reactions will start to instantiate the workflow, as presented in Section 3.2. After the instantiation, a number of reactions will be launched in succession to execute this service composition, as we will present later in Section 3.3 and Section 3.4.

3.1.3 Interactions between Chemical Services

In HOCL implementation, a chemical solution is not a closed space. The *put* primitive is defined to write a (number of) molecule(s) to a remote chemical solution. It requires two parameters: 1) the signature of the chemical solution where you want to pass the molecules and 2) the molecules that are expected to pass. Details about the implementation of the *put* primitive can be found in Section 4.2.

Therefore, by calling the *put* primitive, the chemical rule *send* is defined in the middleware to automate the interaction between chemical services: when a molecule of a specific pattern is produced, it will be automatically sent to a remote multiset by certain rules. As defined in Program 3.4, once some molecules (noted by w) are wanted to be sent to another solution *dest*, w has to be packaged into a *"To_send"* tuple, noted as follows:

$$\text{“}To\_send\text{”} : dest :< w >$$

Program 3.3: The Description of a New SBA Instance

```
"Instance":<
  "Instance_id":"CCS-BestGarage001",
  "Task":"t1":"diagnostic":"Abstract":<
    "In":"Start":<Null>,"Out":"XOR-split":<"t2","t3">>:Null,
  "Task":"t2":"reparation":"Concrete":<
    "In":"Seq":<"t1">,"Out":"Seq":<"t4">>:"CWS-RepManager",
  "Task":"t3":"reparation":"Concrete":<
    "In":"Seq":<"t1">,"Out":"Seq":<"t4">>:"CWS-fastRepairCar",
  "Task":"t4":"billing":"Abstract":<
    "In":"XOR-join":<"t2","t3">,"Out":"END":<Null>>::Null,
  ARs, IRs, ARS, InstRs,
  "Data":<"Car":<"Type":"RARE", "ID":"ABCDEF">>
>
```

Program 3.4: The Definition of the Reaction Rule *send*

```
let send =
  replace "To send":dest:<?w>, ?l
  by l
  if put(dest, w)
```

The emergence of such a *"To_send"* tuple in the local solution will activate the rule *send*, which calls the *put* primitive to write the molecule w to the chemical solution *dest*.

Figure 3.3 presents a concrete example to illustrate the execution of the rule *send*. cws_1 and cws_2 are two chemical solutions, which may be running on different machines and cws_1 contains the rule *send*. During the reactions in the solution of cws_1, a "To_send" tuple is generated by another reaction rule which indicates to pass a string "message!" to the remote solution cws_2. And then, the rule *send* is activated, which consumes the "To_send" tuple and transfers the string to the remote solution cws_2. Please note that for the reason of simplicity, we only use the solution name instead of the full signature of a chemical solution in the rest of this dissertation.

3.2 Context-Aware Service Selection

SBA providers can define the workflow either in a concrete or an abstract manner. The former approach requires an SBA provider to directly specify a service composition which can deliver the expected functionality by means of service collaboration. By contrast, for the abstract approach, an SBA provider only describes the workflow in an abstract way, at design time, by coordinating a set of abstract tasks with specific functional requirements. The abstract workflow cannot be executed since each task lacks the binding reference so that the required functionality of each task cannot be delivered. Therefore, the workflow is required to be instantiated at runtime before the execution can start.

The instantiation of workflow refers to the selection of a suitable service for each task from a number of candidates which can provide the same functionality[2] but differ in Quality

[2]As we stated in the previous chapters, the interface mediation is not considered in this dissertation. Hence, two services capable of providing the same functionality refers to two services with the same interface.

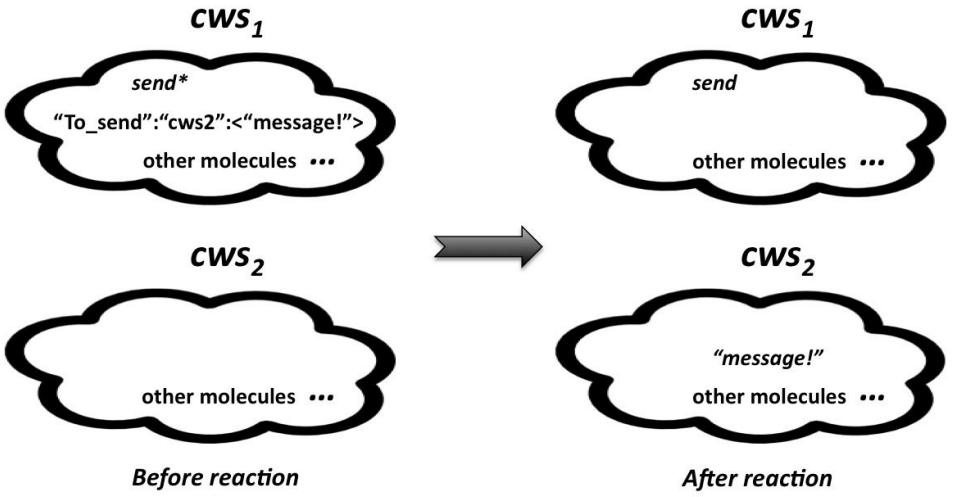


Figure 3.3: Illustration: Movement of Molecule

of Services (QoS). Different combinations of services can result in different compositions of services, followed by the same workflow structure. Two distinct service compositions may lead to different end-to-end QoS of SBA. Therefore, compared to concretely defined workflows, dynamic instantiation of abstract workflows provides more flexibility in building SBAs.

However, dynamic instantiation also brings additional challenges, as we have discussed in Section 1.2. In this section, we introduce how to select suitable services at runtime by means of chemical reactions in the middleware. Section 3.2.1 addresses service selection based on local constraints and Section 3.2.2 discusses the global service selection problem.

3.2.1 Service Selection Based on Local Constraints

In this section, we use the "Best Garage" as an illustrative example to demonstrate service selection based on local constraints. In this example, two tasks are concretely predefined (task t_2 and t_3): the binding information for both tasks are explicitly provided by SBA providers, as described in Program 3.1. However, task t_1 and t_4 have to bind to suitable services according to the specific execution context. The selection of service for each task has to follow some local constraints which can be determined by either the business policies or the customer's preferences.

3.2.1.1 Business policy driven service selection

Sometimes, business policies provide the guidelines to decide how to select services in a specific execution context. We use the following scenario to illustrate business policy driven service selection.

Selection scenario 1: choose a suitable technician. Task t_1 can bind to either a senior or an assistant technician service. The selection of a technician follows the following rules: 1) for an ordinary car, an assistant is invited for the diagnosis; 2) a rare-type car (such as a vintage car or a sports car) has to be diagnosed by a senior technician.

In the middleware, each business policy can be expressed by a (set of) reaction rule(s).

Program 3.5: Chemical Rules for Local Service Selection

```
% Chemical Rules for Business Policy Driven Service Selection
let bindT1ToExpert =
  replace-one "Data":<"Car":<"Type":"RARE", ?w>,?l>,
     "Task":"t1":"diagnostic":"Abstract":<?k>:Null
  by "Data":<"Car":<"Type":"RARE", w>,l>,
     "Task":"t1":"diagnostic":"Concrete":<k>:"CWS-SeniorT"
let bindT1ToAssistant =
  replace-one "Data":<"Car":<"Type":"ORDINARY", ?w>,?l>,
    "Task":"t1":"diagnostic":"Abstract":<?k>:Null
  by "Data":<"Car":<"Type":"ORDINARY", w>,l>,
     "Task":"t1":"diagnostic":"Concrete":<k>:"CWS-JuniorT"

% Chemical Rule for Customer Preference Driven Service Selection
let prebindT4 =
  replace-one "Task":"t4":"billing":"Abstract":<?k>:Null
  by "Task":"t4":"billing":"Concrete":<k>:"CWS-payByCard"
```

As a proof of concept, we provide two reaction rules in Program 3.5 that implement the business policies used in *selection scenario 1*. First of all, the rule *bindT1ToExpert* describes the policy for binding the senior technician service. We suppose that the "Best Garage" has a list that classifies all the models of cars into two different types: *ordinary* cars and *rare* cars. Accordingly, the type of a car can be determined when a client comes to the garage, and it is provided as an initial parameter to create an SBA instance (it can be found in the "Data" sub-solution of an SBA instance). Therefore, the rule *bindT1ToExpert* checks the type of the car and binds the abstract task t_1 to *CWS-SeniorT* if the type of the car is classified as "*RARE*". Meanwhile, the runtime state of task t_1 is changed from "Abstract" to "Concrete". On the other hand, the rule *bindT1ToAssistant* is similarly defined to bind task t_1 to *CWS-JuniorT* for an ordinary car. Please note that both rules are defined as one-shot rule.

3.2.1.2 Customer preference driven service selection

However, sometimes the selection of service is determined by the customer's preferences. Consider the following scenario.

Selection scenario 2: choose the mode of payment. The customer has multiple possible ways to pay the bill. As a proof of concept, we only consider two of them, namely to pay directly by credit card or to pay by check. The selection of the payment mode depends on the customer's preference.

However, the customer's preference is sometimes unavailable at the beginning of the execution. For example, the customer can select the mode of payment only when (s)he is asked to pay the bill of cost (the execution of workflow arrives to the task t_4). Moreover, even though the preference is provided at the beginning, it still can be changed during the execution. As a result, in the middleware, the task always pre-binds to the service that is most likely to be selected according to the past executions at the beginning of the execution. Later, when the execution arrives to that task, if the customer provides a different preference, some adaptation rules can be simply applied to changed the binding reference (introduced later in Section 3.3.3). As an example, in Program 3.5, the rule

Program 3.6: Description of a Concrete Workflow for the "Best Garage" Example

```
"Instance":<
  "Instance_id":"CCS-BestGarage001",
  "Task":"t1":"diagnostic":"Concrete":<
      "In":"Start":<Null>,"Out":"XOR-split":<"t2","t3">>:"CWS-SeniorT",
  "Task":"t2":"reparation":"Concrete":<
      "in":"Seq":<"t1">,"Out":"Seq":<"t4">>:"CWS-RepManager",
  "Task":"t3":"reparation":"Concrete":<
      "In":"Seq":<"t1">,"Out":"Seq":<"t4">>:"CWS-fastRepairCar",
  "Task":"t4":"billing":"Concrete":<
      "In":"XOR-join":<"t2","t3">,"Out":"End":<Null>>:"CWS-payByCard"
  "Data":<"Car":<"Type":"RARE", "ID":"ABCDEF">>
>
```

prebindT4 is defined to pre-bind task t_4 to *CWS-payByCard*, since most of customers have finally payed directly using the credit card in the past. By applying the rule *prebindT4*, task t_4 becomes a concrete task.

By executing reaction rules for local service selection, the sample SBA instance provided in Program 3.3 can be fully instantiated. As a proof-of-concept example, we assume an execution context for repairing a sport car, the context-aware service selection finally lead to the construction of the sample concrete workflow described in Figure 3.6. This sample workflow will be used in the rest of our discussions to illustrate our approaches, for example, the coordination models for executing a service composition.

3.2.2 Global Service Selection

However, sometimes service selection is required to be performed with the consideration of global constraints. A *global constraint* describes the (QoS) restriction in service selection that involves multiple (or all the) tasks. For example, the services selected for two consecutive tasks have to be connectable, which means they should have the same (or similar) interface definition so that they can talk to each other. Furthermore, sometimes, the selection of services also has to satisfy the aggregated QoS of a sub-graph of workflow (e.g., in the medical-care context, some critical tasks are expected to be finished within a limited time duration) or even the global QoS over the entire SBA (e.g. the customer expects to get the response within a certain duration).

In our illustrative example, global service selection is not needed, since the time and cost for the reparation is estimated by the SBA provider according to the preliminary diagnostic of the car. The customer is not allowed to impose any QoS constraints on the entire reparation process, because (s)he lacks professional knowledge. However, our approach is able to be extended to express service selection with the consideration of global constraints. In this section, we present how to model global service selection in terms of chemical reactions.

3.2.2.1 Problem Definition

Most of approaches proposed in the literature assume that a provider can deliver a service on only one quality level. In this case, a service provider is selected from a *service class*, defined as a group of service providers that can deliver the same functionality. However, in the reality, a service provider is able to provide its service on multiple QoS levels in

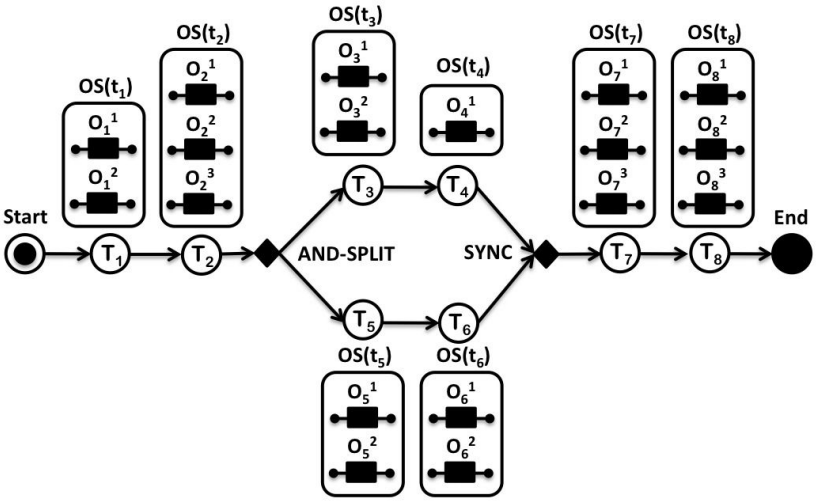


Figure 3.4: Offer Sets

order to meet the requirements of different customers. Take the telecommunication service providers for example, when you want to sign a mobile phone contract, you might have multiple choices for the providers (Vodafone, Orange, etc.) and each provider may offer you multiple service packages with different QoS and cost.

We borrow the concept of *offer* from the example of telecommunication providers. In the middleware, each service provider can define multiple offers to advertise its services on different levels. An *offer* describes both functional and non-functional aspects of a service delivery as well as the identification information of the related service provider, denoted by an offer tuple:

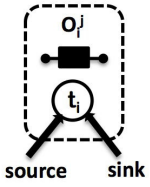
$$\text{“Offer”}:cwsi:fi:<\text{“QoS”}:\text{qi}>$$

An offer tuple is composed of four parts: 1) a constant string “Offer” as a declaration of an offer tuple; 2) *cwsi* is the signature of the related chemical service which defines this offer; 3) *fi* is the functionality that *cwsi* is expected to provide; 4) finally, a sub-solution is used to encapsulate a number of QoS expectations. As a proof of concept, we only consider the expected response time and cost in our discussion.

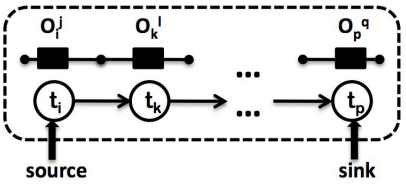
By this means, the service selection problem is transformed to the selection of appropriate offers: a service is selected if and only if one of its offers is selected by a service composition. An *offer set* (OS) is defined as $OS_i = \{o_i^j | 1 \leq j \leq M_i\}$, where o_i^j represents the j^{th} offer that can execute task t_i and M_i is the number of offers in OS_i, as depicted in Figure 3.4. A concrete workflow is constructed by binding each activity t_i to an appropriate offer in the related offer set OS_i, denoted as $cwf = \{t_i \leftarrow o_i^{n_i} | 1 \leq i \leq n; 1 \leq n_i \leq M_i\}$. By introducing the concept of offer, our approach is also applicable to the case that a provider can deliver a service on more than one quality levels.

3.2.2.2 Partially Instantiated Workflow (PIW).

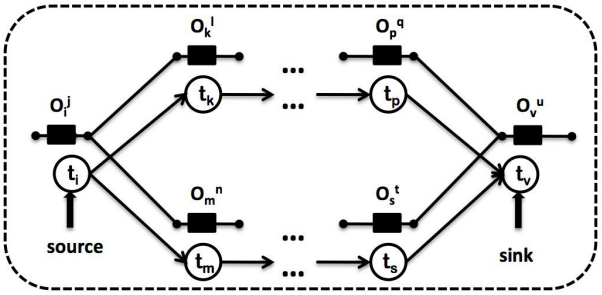
Before discussing how to perform global service selection using chemical reactions, we first introduce the concept of Partially Instantiated Workflow (PIW). A PIW is a *structured* sub-graph of a concrete workflow, which has to meet the following requirements: 1) each PIW has only one source task (the entry point) and one sink task (the exit point); 2) If the source task is distinct from the sink task, all the successors (precedents) of the source



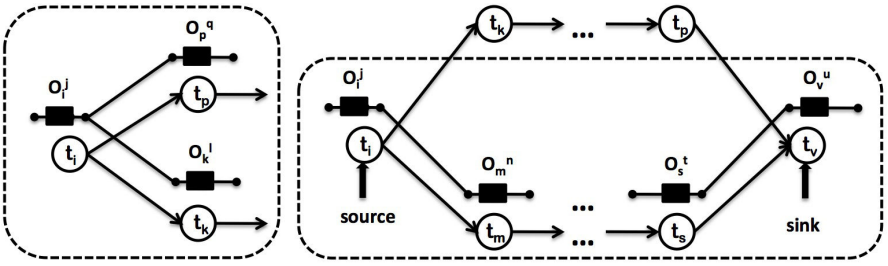
(a) IT-PIW: Instantiated Task



(b) IC-PIW: Chain of Instantiated Tasks



(c) ISB-PIW: Instantiated Structured Block



(d) Counter Example 1

(e) Counter Example 2

Figure 3.5: Partially Instantiated Workflow (PIW)

(sink) task belong to the PIW. 3) For all the other tasks except the source and sink tasks, their precedent(s) and successor(s) belong to this PIW. 4) Each task in a PIW is a concrete task with binding reference.

According to this definition, a PIW can be classified into four categories:

- **Instantiated Task (IT-PIW).** First of all, an IT-PIW refers to single concrete task. As illustrated in Figure 3.5(a), an instantiated task t_i can be seen the basic PIW that acts as the building block of other complex PIWs. Task t_i is the source task as well as the sink task of this PIW.
- **Instantiated Chain (IC-PIW).** Then, a chain of instantiated tasks is also a PIW. As depicted in Figure 3.5(b), an instantiated chain presents a line of instantiated tasks $\{t_i \rightarrow ... \rightarrow t_p\}$, which are expected to be executed in sequential order.
- **Instantiated Structured Block (ISB-PIW).** An ISB-PIW presents a more complicated PIW. As illustrated in Figure 3.5(c), an ISB PIW is sub-workflow composed of several ICs/ITs. This sub-workflow has to meet the above-mentioned 4 requirements. Figure 3.5 also provides some counter examples. Neither the instantiated

sub-workflow in Figure 3.5(d) nor the one in Figure 3.5(e) is an ISB-PIW. The former workflow violates the second requirement since it has two sink tasks. The latter workflow violates the third requirement, since t_k is one of the successors of t_i but it is not included in the sub-workflow.

- **Fully Instantiated Workflow (FIW).** A FIW is a special PIW that refers to a fully instantiated workflow, in other words, every task in the workflow has bound to a selected constituent service[3]. The source task of a FIW equals to the first task of the workflow and the sink task is the last task of the workflow.

Using the chemical metaphor, a PIW is expressed as a complex molecule, which is defined by a PIW tuple:

$$\text{“PIW”}:<\text{“Source”}:\text{ti},\text{“Sink”}:\text{tj},\text{“QoS”}:<?\text{qos\_piw}>,\textit{a list of instantiate task tupes}>$$

A PIW tuple starts with a constant string “PIW” and followed by a solution which includes the information of a PIW: 1) the identity of the source task; 2) the identity of the sink task; 3) the aggregated QoS of this PIW described by a “QoS” sub-solution; 4) a list of instantiated task tuples. And an instantiated task extends a task tuple with the QoS expectations, denoted as:

$$\textit{“IT”}:\textit{task\_id}:\textit{func}:\textit{stat}:<\text{“QoS”}:<?\textit{qos}>,?\text{neighbors}>:\textit{cws\_sol}$$

As a concrete example, the PIW in Figure 3.5(a) can be expressed as follows:

“PIW”:<
“Source”:“ti”,“Sink”:“ti”,“QoS”:<“Time”:5.0, “Price”:0.1>,
“IT”:“ti”:“fi”:“Concrete”:<
“In”:wfp_in:<?*in* >,“Out”:wfp_o:<?*out* >,“QoS”:<“Time”:5.0,“Price”:0.1>
>:“CWSi”
>

with the offer o_i^j in Figure 3.5(a) defined as:

$$\text{“Offer”}:\text{“CWSi”}:\text{“fi”}:<\text{“Time”}:5.0,\ \text{“Price”}:0.1>$$

3.2.2.3 Chemical Reactions for Global Service Selection

The instantiation of workflow with the consideration of global constraints is modeled as a *recursive process* of building PIWs: from the construction of a set of simple ITs to complicated ICs and ISBs, until the entire workflow is fully instantiated. The instantiation completes when a FIW is identified.

Construction of instantiated tasks (IT-PIWs). The instantiation process starts with local offer elimination. The objective is to remove the offers that cannot meet the local (QoS) constraints. For example, SBA provider may want a specific task to complete within a certain time duration. In this case, all offers with longer expected response time will be removed. As an example, the reaction rule *eliminateOfferForTaskTi* defined in Figure 3.7 removes all the offers for task t_i (with the required functionality f_i) with the

[3]Although the word *fully instantiated* and *partially instantiated* are a little contradicting, but both of them follow the same molecular description so that we defined FIW as a *special* type of PIW.

Program 3.7: Reaction Rules for Generating Instantiated Task (IT-PIW)

```
1 % Reaction rule for local elimination
2 let eliminateOfferForTaskTi =
3   replace "Offer":cws:fi:<"Time":t,?w>,
4     "Task":ti:f_req:"Abstract":<?k>:Null
5   by "Task":ti:f_req:"Abstract":<?k>:Null
6   if f_req.equals(fi) && t>5.0
7
8 % Reaction rule for generating IT-PIW
9 let instantiateTask =
10  replace "Offer":cws:fi:<?w>,
11    "Task":id_task:f_req:"Abstract":<?k>:Null
12  by "Task":id_task:f_req:"Abstract":<k>:Null,
13    "IT":id_task:f_req:"Concrete":<k,"QoS":<w>>:cws
14  if f_req.equals(fi)
15 let generateIT-PIW =
16  replace "IT":id_task:f_req:"Concrete":<k,"QoS":<w>>:cws
17    "PIW":<
18      "Source":id_task, "Sink":id_task, "QoS":<w>,
19      "IT":id_task:f_req:"Concrete":<k,"QoS":<w>>:cws
20    >
```

expected response time longer than 5 seconds. These chemical reactions consume (remove) all the unsuitable offers.

After the local elimination of offers, all local constraints can be satisfied and then we can consider how to recursively build PIWs in order to construct a FIW. The first step is to generate a number of basic IT-PIWs based on different possibilities for instantiating a task. For each task in the workflow, the rule *instantiateTask* defined in Figure 3.7 is applied to generate an instantiated task tuple based on each offer for this task (L.13). Then, the rule *generateIT-PIW* is activated to construct an IT-PIW based on each instantiated task. Both source task and sink tasks of this PIW point to this task (L.18). The aggregated QoS of this IT PIW equals to the expected QoS of this instantiated task (represented by w in L.16). After the first-round reactions, all qualified offers are transformed to a number of IT-PIWs.

Construction of instantiated chains (IC-PIWs). In the following, the instantiation of workflow is performed by sequentially connecting PIWs. First, two connectable IT-PIWs can be aggregated to construct an instantiated chain (IC-PIW); and then, smaller IC-PIWs can be aggregated along an execution path to form bigger ones. The construction of IC-PIW is performed as cell fusion process, as illustrated in Figure 3.6. Each PIW can be seen as a cell, which encapsulates a number of molecules to express an instantiated sub-graph of workflow. If the sink task of a PIW (noted as PIW_1) and the source task of another PIW (noted as PIW_2) are sequentially connected, two PIWs will merge into a bigger PIW cell which represents a longer instantiated chain. In this context, the source task of the new PIW is the source task of PIW_1, and the sink task of the new PIW is the sink task of PIW_2.

The rule *generateInstChain* defined in Program 3.8 implements the aggregation of two instantiated chains. A bigger IC-PIW (L.10-16) is constructed by merging the contents of two connectable smaller PIWs (L.02-05 and L.06-09). All the instantiated tasks from both PIWs are copied into the new PIW, and the aggregated QoS of the new PIW is also

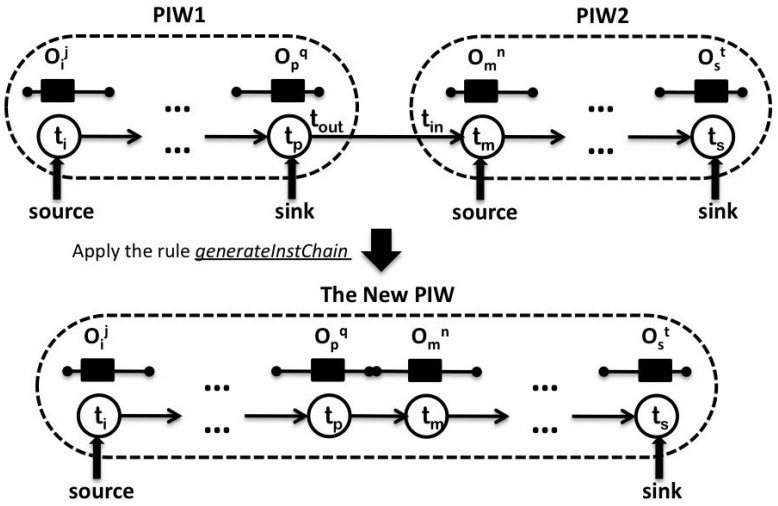


Figure 3.6: Illustration: Generation of a New Instantiated Chain (IC-PIW)

Program 3.8: Reaction Rule for Generating Instantiated Chain (IC-PIW)

```
1  let generateInstChain =
2    replace "PIW":<
3        "IT":t_p:f_p:"Concrete":<"Out":"Seq":<t_out>,?w1>:cws_p,
4        "Source":t_i,"Sink":t_p,"QoS":<"Time":tm1,"Price":pr1>,?l1
5      >,
6      "PIW":<
7        "IT":t_m:f_m:"Concrete":<"In":"Seq":<t_in>,?w2>:cws_m,
8        "Source":t_m,"Sink":t_s,"QoS":<"Time":tm2,"Price":pr2>,?l2
9      >
10   by "PIW":<
11       "Source":t_i, "Sink":t_s,
12       "IT":t_p:f_p:"Concrete":<"Out":t_out,?w1>:cws_p,
13       "IT":t_m:f_m:"Concrete":<"In":t_in,?w2>:cws_m,
14       "QoS":<"Time":(tm1+tm2),"Price":(pr1+pr2)>,
15       l1,l2
16     >
17   if t_out.equals(t_m) && t_in.equals(t_p) && Cond_g
```

calculated. In this case, both the response time and cost are aggregated based on the addition function.

This reaction is based on the following conditions (L.17): 1) both chains have to be sequentially connected. The first PIW has only one successor which points to the source task of the second PIW, and the second PIW has only one precedent which points to the the sink task of the first PIW. 2) Some global constraints (optional), noted by *Cond_g*, have to be satisfied if it is defined. As we have introduced in the beginning of this section, the expression of the global constraints *Cond_g* varies according to the execution contexts. Please note that *Cond_g* can be a compound conditional statement which expresses two or more conditions to be tested.

In the following, we provide a concrete example to express *Cond_g*. Considering the context of arrangement of professors to give courses to the students in a college, each professors proposes several available time intervals for giving his/her courses. In this context, each available intervals can be seen as an offer, which specifies the expected

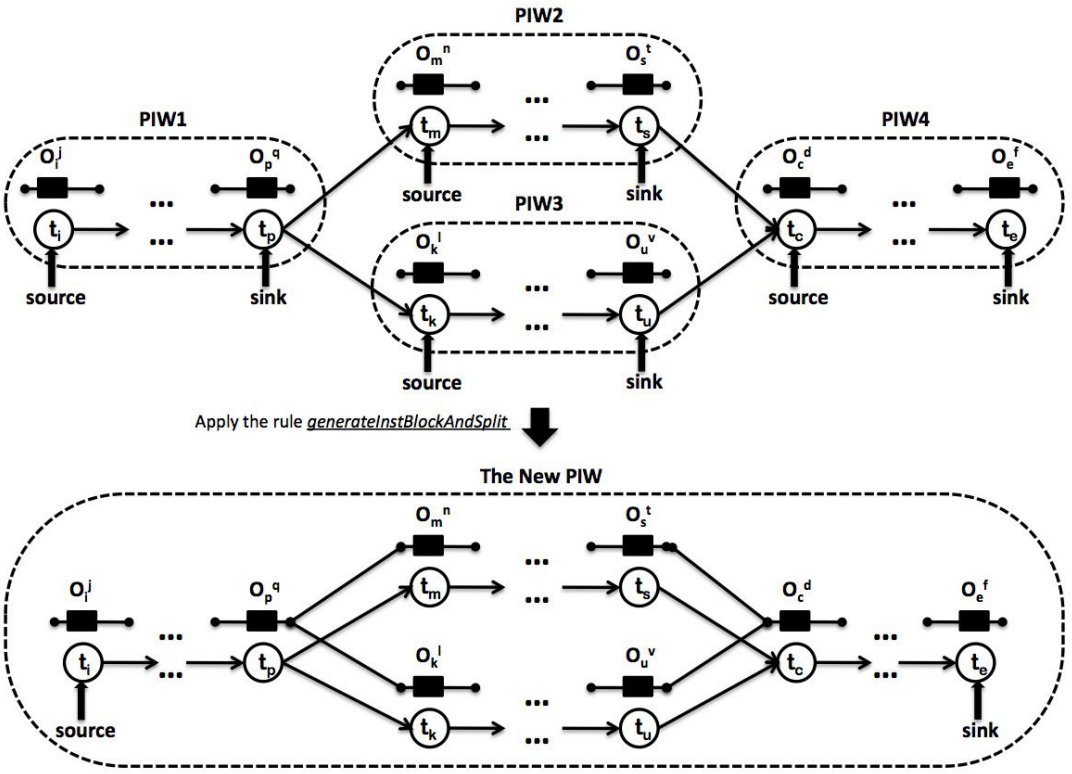


Figure 3.7: Illustration: Generation of a New Instantiated Block (ISB-PIW)

duration by which a service (course) can be delivered. Each offer has two QoS attributes: "Start_time":*tm_s* and "Terminate_time":*tm_t*, indicating respectively the expected start time and termination time. Therefore, two services can be connected if and only if the termination time of the first offer is earlier than the start time of the second one. In this case, the *Cond_g* can be expressed as: *tm_t1*.earlyThan(*tm_s2*).

Construction of instantiated structured blocks (ISB-PIW). Having presented the instantiation of a sequential executed tasks, in the following, we will discuss how PIWs can be aggregated over complex workflow patterns to construct instantiated structured block (ISB-PIW). A structured block is a subgraph of workflow where 1) each split node is associated with exactly one join node and vice versa and 2) each path in the workflow graph originating in a split node leads to its corresponding join node. As an example, Figure 3.7 depicts four PIWs, noted respectively as PIW_1, PIW_2, PIW_3 and PIW_4. The sink task of PIW_1 connects to the source tasks of PIW_2 and PIW_3; on the other side, the sink tasks of both PIW_2 and PIW_3 connect to the source task of PIW_4. In this case, all four PIWs can be merged into a bigger structured instantiated block. The source task of the new PIW is the source task of PIW_1 and the sink task of the new PIW is the sink task of PIW_4.

Suppose that after the execution of PIW_1, both PIW_2 and PIW_3 in Figure 3.7 will be executed in parallel. Then, the executions of two PIWs have to be synchronized before executing PIW_4. The rule *generateISBAndSplitSync* defined in Program 3.9 implements the construction of an ISB-PIW in this context. It requires four PIW tuples as the input molecules. Since two branches are executed in parallel, the aggregated execution time depends on the PIW with longer expected execution time (L.20). However, the aggregated

Program 3.9: Reaction Rules for Generating AND-Split Instantiated Block (ISB-PIW)

```
let generateISBAndSplitSync =
  replace "PIW":<
    "IT":t_p:f_p:"Concrete":<"Out":"AND-Split":<t_out1,t_out2>,?w1>:cws1:<?
        qos1>,
    "Source":t_i,"Sink":t_p,"QoS":<"Time":tm1,"Price":pr1>,?l1
    >,
    "PIW":<
      "Source":t_m,"Sink":t_s,"QoS":<"Time":tm2,"Price":pr2>,?l2
    >,
    "PIW":<
      "Source":t_k,"Sink":t_u,"QoS":<"Time":tm3,"Price":pr3>,?l3
    >,
    "PIW":<
      "IT":t_c:f_c:"Concrete":<"In":"Sync":<t_in1,t_in2>,?w4>:cws4:<?qos4>,
      "Source":t_c,"Sink":t_e,"QoS":<"Time":tm4,"Price":pr4>,?l4
    >
  by "PIW":<
      "Source":t_i, "Sink":t_e,
      "IT":t_p:f_p:"Concrete":<"Out":"AND-Split":<t_out1,t_out2>,?w1>:cws1
          :<?qos1>,
      "IT":t_c:f_c:"Concrete":<"In":"Sync":<t_in1,t_in2>,?w4>:cws4:<?qos4>,
      "QoS":<"Time":(tm1+tm4+MAX(tm2,tm3)),"Price":(pr1+pr2+pr3+pr4)>,
      l1,l2,l3,l4
    >
  if t_out1.equals(t_m)&&t_out2.equals(t_k)&&t_in1.equals(t_s)&&t_in2.equals
      (t_u)&&Cond_g
```

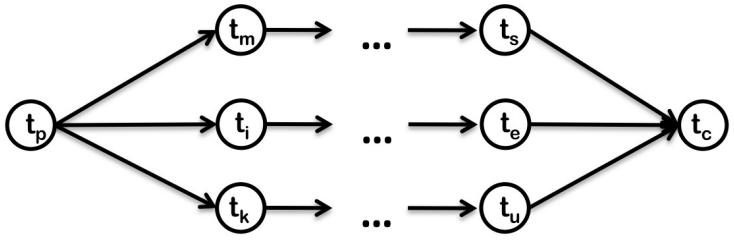
price is still the sum price of all four PIWs. Please note that the global constraints $Cond\_g$ can be similarly expressed as we have introduced before.

As a proof of concept, we have only provided the reaction rules for generating ISB-PIW based on *and-split* and *synchronization* workflow patterns. However, other combination of different workflow patterns for the split and join node can be also expressed using reaction rules in the similar way (e.g. XOR-Split and Simple-Merge, etc). The difference lies in the fact that different functions are used for calculating the aggregated QoS of the new PIW, as we have presented in section 1.1.3.

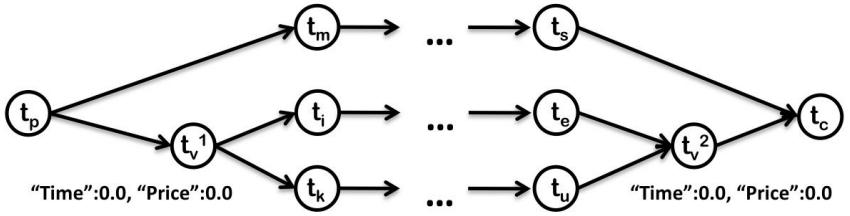
3.2.2.4 Workflow Transformation.

In the above-introduced example, the rule *generateISBAndSplitSync* can only be applied to construct an ISB with two branches, since it explicitly specifies that the sink task of the first PIW (defined as t_p) requires two successors, namely as t_{out1} and t_{out2} (L.03). However, SBA provider may define a workflow with more than two parallel or exclusive execution branches. For example, the workflow in Figure 3.8(a) cannot activate the rule *generateISBAndSplitSync* since it comprises three parallel execution branches. In this case, the SBA provider has to define new rules to instantiate a workflow with an any number of branches. The definitions of these rules are similar to the rule *generateISBAndSplitSync*, except that they require more PIWs as input molecules.

Our objective is to provide a set of *generic* rules that can be used to instantiate all types of workflow with minimum effort of SBA providers. For this purpose, we are going to introduce workflow transformation which is able to transform any structured workflow to a *standard structured workflow*. A standard structured workflow is a sub-set of structured



(a) Original Workflow (Before the Transformation)



(b) Transformed Workflow (After the Transformation)

Figure 3.8: Illustration: Workflow Transformation

workflow where each split/join node can only have two successors/precedents.

The objective of workflow transformation is to ensure that each split/join node has only two successors/precedents so that the rule *generateISBAndSplitSync* can be used in generic cases. To achieve that, new virtual tasks are created and added into the workflow. A *virtual task* can be seen as a place holder in the workflow that has no functional nor non-functional properties. In order to transform the workflow in Figure 3.8(a) into a standard structured workflow, two virtual task (t_v^1 and t_v^2) are created and added into the workflow. As illustrated in Figure 3.8(b), t_v^1 is a successor of task t_p and the precedent of both task t_i and t_k while t_v^2 is a successor of task t_e, t_u and the precedent of task t_c. The expected time and cost of t_v^1 and t_v^2 are all 0.0. For the workflow with more than three branches, the workflow transformation can be performed similarly in a recursive way. By this means, the original workflow can be transformed to a standard structured workflow.

In the following, all the instantiation rules can be applied to construct a fully instantiated workflow. To instantiate the workflow presented in Figure 3.8(b), a number of IT-PIWs are firstly constructed; Then several IC-PIWs are built by aggregating IT-PIWs along all three execution chains: $\{t_m,...,t_s\}$, $\{t_i,...,t_e\}$ and $\{t_k,...,t_u\}$. Later, the rule *generateISBAndSplitSync* is applied to construct the ISB-PIW for the subgraph of workflow $\{t_v^1,...,t_v^2\}$. And finally the rule *generateISBAndSplitSync* is used again to build a fully instantiated workflow.

3.2.2.5 Aggregated/Global QoS Verification.

Sometimes, global service selection is expected to meet the constraints on the aggregated QoS of a subgraph of workflow or the entire workflow. Take the workflow in Figure 3.8 for example, suppose that the SBA provider expects the execution of the subgraph of workflow $\{t_m, \ldots, t_s\}$ to be finished within 15 seconds at the cost of less than 0.8 euros. The rule defined in Program 3.10 describes the verification of such global constraints. It reacts with any a FIW that starts with the task t_m and ends with the task t_s. After verifying the

Program 3.10: Reaction Rules for Global QoS Verification

```
1 let verifyAggQoSConstraint1 =
2   replace "PIW":<"Source":"t_m","Sink":"t_s","QoS":<"Time":t,"Price":pr>, l>
3   by l
4   if t>15 || pr>0.8
```

aggregated QoS of this FIW, if the expected execution time is greater than 15s or the cost is over 0.8 euros, this FIW tuple will be consumed and all the instantiated tasks included in it will be released (represented by l in L.03).

We suppose that for each task in the workflow, multiple IT-PIWs can be constructed (multiple offers can provide the expected functionality requirement of each task). In the following, if a PIW for the subgraph of workflow $\{t_i, \ldots, t_j\}$ can meet the expected QoS, it can be used later to construct the fully instantiated workflow. To the contrary, if it cannot meet the required QoS requirements, all the instantiated tasks will be released and then, the instantiation rule introduced in Figure 3.7 will be re-applied to re-generate a number of IT PIWs. And all these "new" (fresh) IT-PIWs will be recombined to build (different) IC-PIWs and ISB-PIW to instantiated the subgraph of workflow $\{t_i, \ldots, t_j\}$. Then, the rule *verifyAggQoSConstraint1* will be used again to check the aggregated QoS of all new PIWs for the subgraph of workflow $\{t_i, \ldots, t_j\}$. In this case, the verification of aggregated QoS is always performed until 1) all PIWs for this subgraph of workflow can meet the QoS requirements or 2) an fully instantiate has been constructed.

3.2.2.6 Discussion.

In the middleware, the instantiation process is performed by a series of continuous reactions that never stop until a fully instantiated workflow is constructed. This approach exhibits some advantages as well as limitations. First of all, traditional approaches for service selection rely on building optimization models or heuristic algorithms, which require SBA providers to develop extra tools that are able to transform a BPEL description of workflow to some mathematical models or vice versa. In chemistry-inspired middleware, no transformation is required so that both design-time and runtime complexity can be reduced. Moreover, the instantiation and execution can be performed in parallel since both processes are described as chemical reactions and different reactions can take place concurrently. Therefore, in some cases (e.g., no constituent service is found for a task), the execution can start before the construction of a FIW completes. Finally, our approach is adaptable to the evolving execution environment. For example, during the instantiation process, if an offer with better QoS is found, it can be dynamically added into the solution and then immediately participate into the instantiation reactions.

However, this approach also presents some limitations (to be solved in the future work). First, the instantiation may not be able to identify a FIW when global constraints are required, even a possible solution exists. This is because the selection of molecules is non-deterministic. Suppose all the PIWs constructed in the first round cannot meet the global QoS constraints according to aggregated QoS Verification. As introduced above, all the instantiated tasks will be released and the second-round reconstruction will start. However, the second-round instantiation may produce the same PIWs as the first round (each rule selects the same input molecules). This may lead to the instantiation into a dead-lock loop that never ends (although it presents a low probability). Moreover, even if a concrete workflow is successfully identified, this non-deterministic selection process

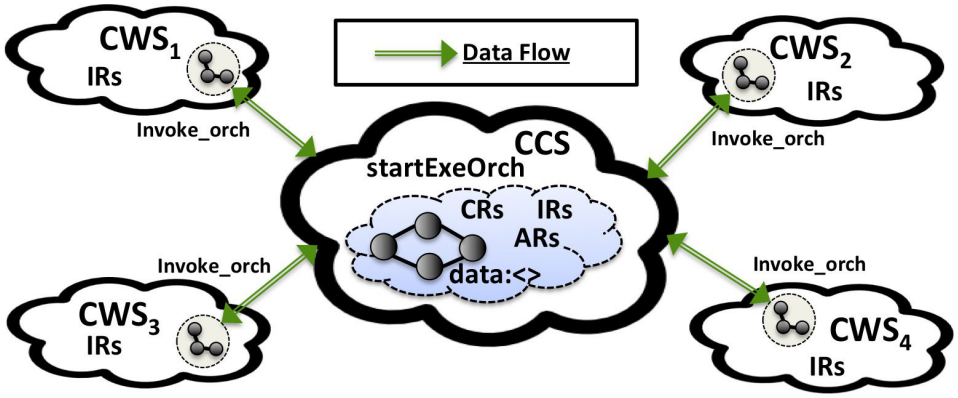


Figure 3.9: Illustration: Orchestration of Chemical Service

may take long time to complete if a lot of global constraints are specified. Finally, the instantiation rules for global selection of service can be used only for structured workflow. PIWs are aggregated based on structured workflow, either a sequential path or a pair of split node and join node.

3.3 Centralized Coordination Model for the Execution of Service Compositions

Up to now, we have presented the creation and the instantiation of a new SBA instance. Then, this SBA instance can be executed by running the instantiated workflow. The execution of a concrete workflow refers to the coordination of all constituent services so that they can cooperate in order to fulfill the ultimate business goal. Both centralized and decentralized coordination models have been introduced in Section 1.3. In this section, we are going to illustrate implementation of the centralized coordination model, known as *service orchestration*, using chemical metaphor in the middleware.

3.3.1 Chemical Service Orchestration Model

In the context of service orchestration, CCS plays the role of the centralized orchestrator, as depicted in Figure 3.9. A number of reaction rules are defined in the solution of CCS (more precisely, in the sub-solution of each running SBA instance) in order to manage the execution of service compositions, for example, to control the execution order of constituent chemical services. As introduced, according to different functionalities, all the reaction rules in the solution of CCS can be classified into three categories, namely the invocation rules (IRs), coordination rules (CRs) and adaptation rules (ARs). In this section, we will introduce a number of CRs and IRs; the ARs will be addressed in Section 3.3.3.

Example of service orchestration. As a proof of concept, we use the "Best Garage" as an example to illustrate the implementation of the orchestration model in the middleware. A possible concrete service composition has been instantiated after the local selection of services, as described in Program 3.6. Once the instantiation process completes, the sub-solution of this SBA instance becomes inert. At this moment, the CR *startExeOrch* in the solution of CCS is activated. As defined in Program 3.11, it modifies the runtime state of the first task, noted by t_s, from "Concrete" to "Executing" (L.06). Using the "Best Garage" example, the task tuple t_s is mapped to t_1; thus the runtime state of t_1 will be

set to "Executing". By this means, some invocation rules in this instance sub-solution will be activated to trigger a series of chemical reactions in the middleware that execute the service composition by the orchestration model.

Program 3.11: Coordination Rules for Service Orchestration

```
let startExeInstOrch =
  replace "Instance":<
    "Task":t_s:f_s:"Concrete":<"In":"Start":<Null>,?w>:cws_s, ?l
    >
  by "Instance":<
    "Task":t_s:f_s:"Executing":<"In":"Start":<Null>,w>:cws_s, l
     >
let correlateResult =
  replace "Reply":inst_id1:cws:<?result>,
    "Instance":<"Instance_id":inst_id2,?w>
  by "Instance":<"Instance_id":inst_id2,"reply":cws:<result>,w>
  if inst_id1.equals(inst_id2)
let completeTask =
  replace "reply":cws:<"Result":<?result>>,"Data":<?d>,
    "Task":ti:fi:"Invoking":<?w>:binding
  by "Task":t_i:f_i:"Executed"<w>:binding,
     "Data":<result,d>
  if cws.equals(binding)
let selectBranchCasePos =
  replace "Data":<"Problem":p::String, ?w>,
    "Task":"t1":"diagnostic":"Executed":<?w1>:binding1::String,
    "Task":"t2":"reparation":"Concrete":<?w2>:binding2::String
  by  "Data":<"Problem":p, w>,
    "Task":"t1":"diagnostic":"Executed":<w1>:binding1,
    "Task":"t2":"reparation":"Executing":<w2>:binding2
  if p.equals("EASY")
let selectBranchCaseNeg =
  replace "Data":<"Problem":p::String, ?w>,
    "Task":"t1":"diagnostic":"Executed":<?w1>:binding1::String,
    "Task":"t3":"reparation":"Concrete":<?w3>:binding3::String
  by  "Data":<"Problem":p, w>,
    "Task":"t1":"diagnostic":"Executed":<w1>:binding1,
    "Task":"t3":"reparation":"Executing":<w3>:binding3
  if p.equals("HARD")
let completeExeOrch =
  replace "Instance":<
    "Task":t_e:f_e:"Executed":<"Out":"End":<Null>,?w>:cws_e, ?l
    >
  by  "Instance":"Completed":<
      "Task":t_e:f_e:"Executed":<"Out":"End":<Null>,w>:cws_e, l
    >
```

1. First of all, the IR *invokeT1* will react to task tuple t_1 with the runtime state of "Executing". As defined in Program 3.12, it prepares an invocation request for executing task t_1, described by an "Invoking" tuple (L.06):

   "Invoking":*cws*:<"operation":*op*, ?*parameters*>

   An invocation request indicates the operation (*op*) of a chemical service (*cws*) to be invoked as well as a number of invocation parameters (name-value pairs).

Program 3.12: Invocation Rules for Service Orchestration (Defined in CCS)

```
1 let invokeT1 =
2   replace "Data":<"Car":<"ID":id, ?w1>, ?w2>
3     "Task":"t1":"diagnostic":"Executing":<?w3>:cws_1,
4   by  "Data":<"Car":<"ID":id, w1>, w2>,
5     "Task":"t1":"diagnostic":"Invoking":<w3>:cws_1,
6     "Invoking":cws_1:<"operation":"diagnose","car_id":id>
7 let invokeT2 =
8   replace "Data":<"Car":<"ID":id,"Time_Exp":q_t,?w1>, ?w2>,
9     "Task":t2:"reparation":"Executing":<?w3>:cws_2
10  by  "Data":<"Car":<"ID":id,"Time_Exp":q_t,w1>, w2>,
11    "Task":t2:"reparation":"Invoking":<w3>:cws_2,
12    "invoking":cws_2:<"operation":"repare","car_id":id,"Time_C":q_t>
13 let invokeT3 =
14  replace "Data":<"Car":<"ID":id,?w1>, ?w2>,
15    "Task":t3:"reparation":"Executing":<?w3>:cws_3
16  by  "Data":<"Car":<"ID":id,w1>, w2>,
17    "Task":t3:"reparation":"Invoking":<w3>:cws_2,
18    "Invoking":cws_3:<"operation":"repare","car_id":id>
19 let invokeT4 =
20  replace "Data":<"payment":<"Card_No":num,"Password":pw>,?w1>, ?w2>,
21    "Task":t4:"billing":"Executing":<?w3>:cws_4
22  by  "Data":<"Car":<w1>, w2>
23    "Task":t4:"billing":"Invoking":<w3>:cws_4,
24    "Invoking":cws_4:<"operation":"payByCard","card_num":num,"password":pw>
25 let prepareInvocationMsg =
26  replace "Invoking":cws:<?parameters>,
27    "Instance_id":inst_id
28  by  "Instance_id":inst_id
29    "To_send":cws:<
30      "Invoke_orch":"CCS-BestGarage":inst_id:<parameters>
31    >
```

2. The generation of an invocation request will activate the rule *prepareInvocationMsg* (defined in Program 3.12) in succession to generate a concrete invocation message for the corresponding chemical service. An invocation message is defined by an "Invoke_orch" tuple (L.30), defined as:

   "Invoke_orch":*requester*:*inst_id*:<"operation":*op*, ?*parameters*>

   It extends the an invocation request with the signature of the CCS and the instance ID (*inst_id*). This invocation message is put into a "To_send" tuple (L.29-31). In our execution context, the following tuple is generated by applying this rule:

   "To_send":"CWS-SeniorT"<
   "Invoke_orch":"CCS-BestGarage":"CCS-BestGarage001":<
   "operation":"diagnose","car_id":id>,
   >

   Then, the rule *send* becomes active and passes the invocation message to the solution of *CWS-SeniorT*.

3. In the following, the execution is continued in the solution of *CWS-SeniorT*. The arrival of an invocation tuple will trigger a series of reactions in its solution. The

rule *invoke_diagnose_orch* defined in Program 3.13 can get the invocation parameters from the invocation tuple (L.03-04), and then interact with the *WS-SeniorT* Web service by calling the *invoke* method defined by the connector (L.07). If the invocation succeeds, the *invoke* method will return a result tuple that packages the computational result, defined as:

$$\text{“Result”}:<?result>$$

Otherwise, in case that a runtime failure arises, the *invoke* method will return an error tuple which contains the detail about the failure, noted as:

$$\text{“Error”}:<?error>$$

Later, the response (either positive or negative) will be packaged in a *“Reply”* tuple (L.07) with the information of the corresponding instance ID as well as the signature of *CWS-SeniorT* (used by *CCS-BestGarage* to know from whom this response comes from). In our execution context, the following reply tuple can be generated:

```
“Reply”:“CCS-BestGarage001”:“CWS-SeniorT”:<
   “Result”:<“Problem”:“EASY”, “Time”:120, “Cost”:50>
>
```

In this case, the evaluation result shows that the car can be directly repair by the “Best Garage” since the problem is easy, the estimated reparation time and cost are respectively 120 hours and 50 €. This tuple is encapsulated in a “To_send” tuple so that it can be returned to *CCS-BestGarage* in succession by the rule *send*.

4. Later, in the solution of *CCS-BestGarage*, the reply from a constituent chemical service will activate the CR *correlateResult*. As defined in Program 3.11, the reply tuple will be passed from the the solution of CCS to the sub-solution of the corresponding SBA instance (L.11).

5. Then, the emergence of a reply tuple in an instance sub-solution will activate the generic CR *completeTask*. As defined in Program 3.11, it sets the runtime state of the corresponding task from “Invoking” to “Executed” (L.16) and then puts the computational result to the “Data” sub-solution (L.17). Please note here, it requires the reply really contains a result tuple rather than an error tuple (L.14). The processing of an error reply will be addressed later in Section 3.3.3.1. In our execution context, the state of t_1 is marked as “Executed”, the diagnosed problem and estimated time and cost are put into the “Data” sub-solution.

6. Once task t_1 has been executed, one of the two CRs will be activated for selecting a succeeding task to continue the execution. As defined in Program 3.11, *selectBranchCasePos* and *selectBranchCaseNeg* express the XOR-split workflow pattern after the execution of task t_1. Firstly, if the the problem is evaluated as “EASY” (L.26), the rule *selectBranchCasePos* forwards the execution to task t_2 by changing the state of t_2 to “Executing” (L.25). On the other hand, the rule *selectBranchCaseNeg* is similarly defined, which marks the state of task t_3 as “Executing” if the problem is considered as “HARD” (L.27-34). For our future explication, we assume the execution is finally forwarded to task t_2.

Program 3.13: Invocation Rules for Service Orchestration (Defined in CWS)

```
//The IR defined in CWS-SeniorT chemical Web service
let invoke_diagnose_orch =
  replace connector,"invoke_orch":requester:inst_id:<
    "operation":"diagnose","car_id":id>
  by  connector,
    "to_send":requester:<
      "Reply":inst_id:"CWS-SeniorT":<connector.invoke("diagnose",id)>
    >

//The IR defined in CWS-RepManager chemical Web service
let invoke_repare_orch =
  replace connector,"invoke_orch":requester:inst_id:<
    "operation":"repare","car_id":id>
  by connector,
    "to_send":requester:<
      "Reply":inst_id:"CWS-SeniorT":<connector.invoke("repare",id)>
    >

//The IR defined in CWS-payByCard chemical Web service
let invoke_payByCard_orch =
  replace connector,"invoke_orch":requester:inst_id:<
    "operation":"payByCard","card_num":num,"pw":pw>
  by connector,
    "to_send":requester:<
      "Reply":inst_id:"CWS-SeniorT":<connector.invoke("payByCard",num,pw)>
    >
```

7. In the following, the IR *invokeT2* defined in Program 3.12 becomes active. Similarly to *invokeT1*, it prepares an invocation message to *CWS-RepManager* for executing task t_2. And then, it marks the runtime state of task t_2 as "Invoking".

8. The invocation message will be sent to the solution of *CWS-RepManager*, where the rule *invoke_repair_orch* is activated to invoke the real Web service "WS-RepManager" and to generate a reply tuple. Its definition is similar to *invoke_diagnose_orch*, as provided in Program 3.13.

9. The reply tuple from *CWS-RepManager* is sent to the solution of *CCS-BestGarage*, and then passed to the sub-solution of the corresponding instance. Then, the runtime state of t_2 is set to "Executed" by the CR *completeTask*. At this moment, the coordination rule *simpleMerge* is activated which expresses the simple-merge workflow pattern before task t_4. The definition of this rule will be introduced later in Section 3.3.2 (refer to Program 3.19). By executing the rule *simpleMerge*, the state of task t_4 is set to "Executing".

10. Similar to previous steps, the invocation rule *invokeT4* is able to generate an invocation message for executing task t_4, which is sent to the solution of *CWS-payByCard*. The rule *invoke_payByCard_orch* in Program 3.13 will interact with the *WS-payByCard* Web service for paying the bill of cost. The result (successful or not) is encapsulated in a reply tuple and finally returned to the sub-solution of this running SBA instance. This result will finally change the state of task t_4 from "Invoking" to "Executed".

Figure 3.10: Workflow Pattern: Sequence

11. When the last task has been executed, the instance tuple is marked as "Completed" by the CR *completeExeOrch* in the solution of *CCS-BestGarage*, as defined in Program 3.11. Up to now, the execution of an SBA instance is completed. The SBA provider can define another set of rule to return the final result to SBA requester. As the limited space, we do not provide the details of these rules.

Discussion. From the above example, we can conclude that the service orchestration is performed by a series of distributed chemical reactions in the middleware. On one side, the solution of a CCS plays the role of the centralized orchestrator. Most of the coordination rules are generic[4] that aim to direct the order of service invocations by manipulating the state of tasks. By contrast, all the invocation rules are specific[5], since each invocation rule has to deals with different invocation parameters thus it can hardly be generic. On the other side, each constituent performs respectively a simple task. Once an invocation tuple arrives to its solution, it invokes the connected Web service. Later, when the result is received, it encapsulates and returns the result to the requesting CCS. All constituent service have no knowledge about the internal logic of SBA and thus they do not participate in coordination of services.

3.3.2 Reaction Rules to Express Complex Workflow Patterns

From the above example, we can see that the coordination rules only manipulate on the state of tasks by deciding which task to be executed next. Once the state of a task is changed to "Executing", a corresponding invocation rule will be activated to prepare and send invocation messages. However, we have only illustrated the expression of exclusive branches in the above example. In this section, we are going to illustrate how to express other complex workflow patterns using reaction rules.

Sequence. The *sequence* pattern refers to the sequential execution of two tasks. As illustrated in Figure 3.10, t_j is the only successor of t_i, and t_i is the only precedent of t_j. Therefore, task t_i and t_j are sequentially executed: the completion of executing task t_i will start the execution of task t_j.

The CR *sequence* defined in Program 3.14 describes the implementation of sequence workflow pattern in terms of chemical reactions. It requires two task tuples as the input molecules, identified respectively as t_i and t_j. t_i has already been executed whereas t_j remains unexecuted (L.02-03). If t_i and t_j are sequentially connected (verified in L.06), the runtime state of t_j is set to "Executing" (L.05), which will later activate the IR *invokeTj* to invoke the corresponding service. By this means, the execution is sequentially handed over from t_i to t_j. Please note that the rule *sequence* is a *generic* rule, that can be used for all sequential execution of tasks in a workflow.

[4]Generic rules can be used for different workflows without any modification (e.g., *startExeInstOrch*). So they can be provided directly by the middleware.

[5]Specific rules describes special policies/knowledges used for a certain SBA (e.g., *invokeT1*). Therefore, they have to be defined directly by the SBA provider.

Program 3.14: The Definition of the Reaction Rule *sequence*

```
1 let sequence =
2   replace "Task":ti:fi:"Executed":<"Out":"Seq":<nextTask>,w1>:bi,
3     "Task":tj:fj:"Concrete":<"In":"Seq":<prevTask>,?w2>:bj
4   by "Task":ti:fi:"Executed":<"Out":"Seq":<nextTask>,w1>:bi,
5     "Task":tj:fj:"Executing":<"In":"Seq":<prevTask>,w2>:bj
6   if tj.equals(nextTask) && ti.equals(prevTask)
```

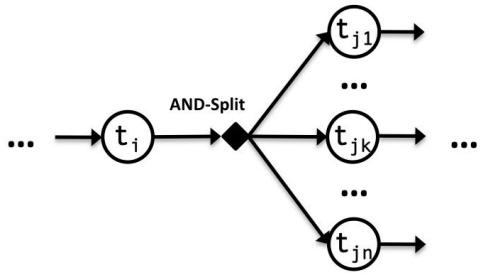


Figure 3.11: Workflow Pattern: And-Split

AND-split. The AND-split pattern refers to the parallel execution of multiple tasks. As illustrated in Figure 3.11, task t_i has n outgoing branches and the k^{th} branch starts with the task t_{jk} ($1 \leq k \leq n$). Under the control of the AND-split pattern, the completion of task t_i will start the execution of all its n successors in parallel.

The rule *and-split* defined in Program 3.15 describes the reactions that model the control of parallel executions of tasks. Each reaction requires two task tuples as input molecules. t_i is an executed task which has a number of outgoing tasks to be executed in parallel and *nextTask* is one of its successors (L.02). t_j is an unexecuted task which has only one precedent *prevTask* (L.03). If t_j is one of successors of t_i (L.06), its runtime state will be changed to "Executing" (L.05). In this case, the execution is continued to t_j. Suppose t_i has n successors, the rule and-split is required to be applied by n times so that the state of all the successors can be changed to "Executing". Similar to the rule *sequence*, the rule *and-split* is also a generic rule that can be applied to all and-split patterns in a workflow.

Exclusive-Choice. Exclusive choice is also known as XOR-split. As illustrated in Figure 3.12, after the completion of task t_i, there are multiple choices to pass the flow. However, only one of its successors can be selected to continue the execution. The selection of each successor is associated with a special condition predefined by the SBA provider.

Suppose that when the condition *condj1* holds, the execution will passed to task t_{j1} after the execution of t_i. As an example, the rule *exclusive-choice-j1* defined in Program 3.16 describes the reaction that selects task t_{j1} to continue the execution. It requires two task tuples, similar as before, t_j is one of the successors of t_i (L.06). If the condition *condj1* is tested to be true, the runtime state of task t_j will be marked as "Executing" (L.05). Different from previous coordination rules, an exclusive choice pattern is expressed by a set of rules. Each rule has to specify the condition under which a specific successor can be executed. Obviously, these rules are not generic. In Section 3.3.1, a concrete example of the exclusive-choice workflow pattern is provided by using the "Best Garage" scenario.

Program 3.15: The Definition of Reaction Rule *and-split*

```
let and-split =
  replace "Task":ti:fi:"Executed":<"Out":"AND-Split":<nextTask,w1>,w2>:bi,
    "Task":tj:fj:"Concrete":<"In":"Seq":<prevTask>,?w3>:bj
  by  "Task":ti:fi:"Executed":<"Out":"AND-Split":<nextTask,w1>,w2>:bi,
    "Task":tj:fj:"Executing":<"In":"Seq":<prevTask>,w3>:bj
  if  tj.equals(nextTask) && ti.equals(prevTask)
```

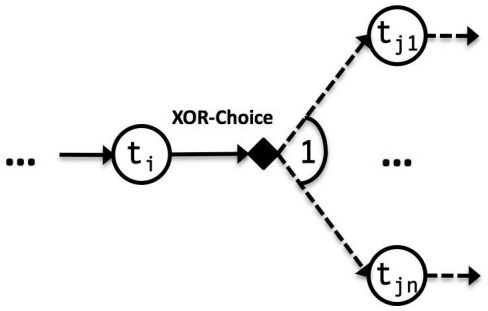


Figure 3.12: Workflow Pattern: Exclusive-Choice

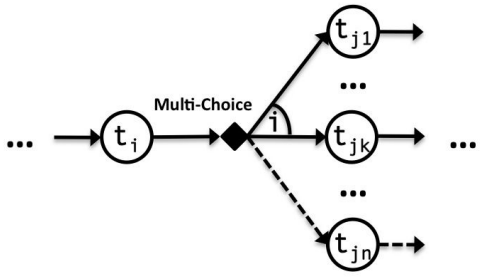


Figure 3.13: Workflow Pattern: Multiple-Choice

Multiple-Choice. Multiple-choice is similar to exclusive-choice, the difference lies in the fact that the execution can be passed to one or multiple outgoing branches. As a concrete example illustrated in Figure 3.13, after the completion of task t_i, if the condition *cond1* holds, the first k tasks (noted as t_{j1}, ... , t_{jk}) will be executed in succession.

The reaction rule *multiple-choice-1* defined in Program 3.17 describes such process as the following chemical reactions. It requires two task tuple t_i and t_j, and t_j is one of the successors of t_i. If the condition *Cond1* is tested to be true, and t_j equals to one of the following tasks, t_{j1}, ..., t_{jk}, then the runtime state of t_j is set to "Executing". Meanwhile, similar rules can be defined to express the mechanisms to select other sets of successors under different conditions. Similar to exclusive-choice, the rules for expressing multiple-choice are neither generic. A specific set of rules are required for each multiple-choice pattern in a workflow.

Synchronization. After discussing some important workflow patterns for the divergence of multiple execution branches, we now present how to express the workflow patterns for the convergence of branches. First of all, the synchronization pattern is used to synchronize multiple execution branches. It is always used together with an AND-split workflow

Program 3.16: The Definition of Reaction Rule *exclusive-choice*

```
1 let exclusive-choice-j1 =
2   replace "Task":ti:fi:"Executed":<"Out":"Exclusive-Choice":<nextTask,?w1>,
        ?w2>:bi,
3     "Task":tj:fj:"Concrete":<"In":"Seq":<prevTask>,?w3>:bj
4   by  "Task":ti:fi:"Executed":<"Out":"Exclusive-Choice":<nextTask,w1>, w2>:
        bi,
5     "Task":tj:fj:"Executing":<"In":"Seq":<prevTask>,w3>:bj
6   if tj.equals(nextTask) && ti.equals(prevTask) && Condj1
```

Program 3.17: The Definition of Reaction Rule *multiple-choice*

```
1 let multiple-choice-1 =
2   replace "Task":ti:fi:"Executed":<
3     "Out":"Multiple-Choice":<nextTask,?w1>, ?w2>:bi,
4     "Task":tj:fj:"Concrete":<"In":"Seq":<prevTask>,?w3>:bj
5   by  "Task":ti:fi:"Executed":<"Out":"AND-Split":<nextTask,w1>, w2>:bi,
6     "Task":tj:fj:"Executing":<"In":"Seq":<prevTask>,w3>:bj
7   if  tj.equals(nextTask) && ti.equals(prevTask) && Cond1 && (tj.equals("tj1
        ")||..||tj.equals("tjk"))
```

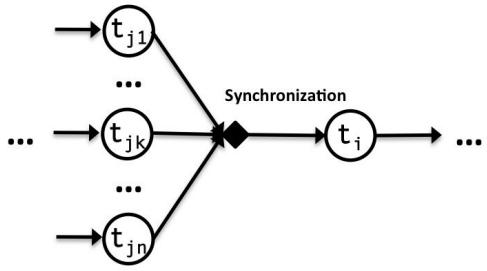


Figure 3.14: Workflow Pattern: Synchronization

pattern. As depicted in Figure 3.14, the execution of task t_j can starts only when all of its precedents have completed the execution.

The synchronization is realized by the cooperation of a pair of reaction rules, as defined in Program 3.18. First of all, the rule *synchronization* requires two task tuples t_i and t_j, and t_i is one of the precedents of t_j. Once t_i is executed, its name is removed from the "In" sub-solution of task t_j. When all the precedents of t_j have completed, the "In" sub-solution becomes empty. At this moment, the second rule *synchronization-done* is automatically activated (as indicated in L.08, it requires an empty "In" sub-solution), which changes the runtime state of t_j to "Executing". The synchronization rules are generic, which can be applied to all synchronization patterns in a workflow.

Simple-Merge. It is also known as *XOR-join*. In contrast to synchronization, the simple-merge pattern provides a means of merging two or more distinct branches without synchronizing them. As illustrated in Figure 3.15, task t_j has a number of precedents. The execution of t_j can start if any precedent task has completed the execution. It is frequently used in conjunction with the exclusive-choice pattern.

The rule *simple-merge* defined in Program 3.19 describes such process. For any two

Program 3.18: The Definition of Reaction Rule *synchronization*

```
1 let synchronization =
2   replace "Task":ti:fi:"Executed":<Out":"Seq":nextTask,?w1>:bi,
3     "Task":tj:fj:"Concrete":< "In":"Sync":<prevTask,?w2>,?w3>:bj
4   by  "Task":ti:fi:"Executed":<"Out":"Seq":nextTask,w1>:bi,
5     "Task":tj:fj:"Concrete":<"In":"Sync":<w2>,w3>:bj
6   if tj.equals(nextTask) && ti.equals(prevTask)
7 let synchronization-done =
8   replace "Task":tj:fj:"Concrete":<"In":"Sync":<>,?w3>:bj
9   by "Task":tj:fj:"Executing":<"In":"Sync":<>,w3>:bj
```

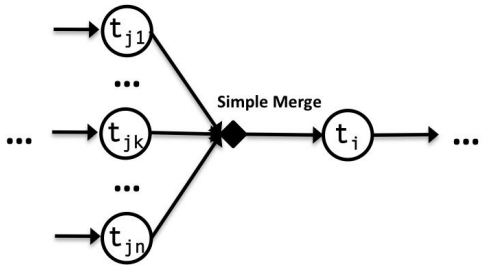


Figure 3.15: Workflow Pattern: Simple Merge

task t_i and t_j, t_i is one of the precedents of t_j. If t_i is executed (L.02) and t_j (L.04) is still unexecuted, the runtime state of t_j is set to "Executing". After the first completion of a precedent, the runtime state of t_j has already changed to "Executing". The completion of other precedents later will not trigger this rule because it requires the convergency task with a state of "Concrete". Therefore, for each simple merge pattern, this rule can be applied only once. The *simple-merge* rule is also a generic rule.

Discussion. From the above presentation, we can conclude that the coordination of service can be *flexibly* expressed using chemical rules in the middleware. First of all, each complex workflow pattern can be easily expressed by simple and few reaction rules. Furthermore, since each split pattern and join pattern are expressed separately, they can be flexibly combined (e.g. AND-split combined with simp-merge). Finally, without structured blocks to express the execution order of services, our approach is also able to express unstructured workflow (arbitrary workflow structures).

3.3.3 Runtime Adaptation of SBA

The execution of SBA involves multiple Web services from different organizations that are executed on distributed computing resources. In such a distributed and loosely coupled execution environment, the execution of an SBA instance may fail, or fail to meet the required quality level. For example, a constituent service may take longer time to respond due to network congestion; moreover, infrastructure failures can cause a service completely responseless. In response to this evolving execution environment, runtime self-adaptation is crucial for service management systems. The adaptive execution reflects the capability to recompose a (part of) workflow on the fly while unexpected failures arise.

In this section, we present runtime adaptation for the service orchestration model. In the middleware, the adaptation can be performed either on the binding level or on the

Program 3.19: The Definition of Reaction Rule *simple-merge*

```
let simple-merge =
  replace "Task":ti:fi:"Executed":<
    "Out":"Seq":<nextTask>,?w1>:bi,
    "Task":tj:fj:"Concrete":<
      "In":"Simple-Merge":<prevTask,?w2>,?w3>:bj
  by  "Task":ti:fi:"Executed":<"Out":"Seq":<nextTask>:bi,
    "Task":tj:fj:"Executing":<"In":"Simple-Merge":<prevTask,w2>,w3>:bj
  if tj.equals(nextTask) && ti.equals(prevTask)
```

workflow level. Section 3.3.3.1 discusses how recover from functional failures by changing binding references. Then, Section 3.3.3.2 presents the adaptation of SBA by modifying the workflow structure in response to non-functional failures.

3.3.3.1 Binding Level Adaption of SBA

Failures are unavoidable during the invocations to constituent services due to the distributed and loosely coupled environment. On one side, all constituent Web services are managed externally by different organizations. The SBA provider lacks the control over the implementations and infrastructures of these third-party Web services. On the other side, using the Internet as the communication medium increases the unreliability. For example, due to the network congestion, an invocation message may be lost so that the corresponding third-party Web service becomes responseless. In this case, the adaptation can be performed on the binding level: the failed task will be re-executed by rebinding to another functional-equivalent chemical service.

The binding-level adaptation for service orchestration is illustrated in Figure 3.16. During the execution of a service orchestration, when an invocation to a Web service WS_i is failed, the corresponding chemical Web service CWS_i will get a reply message containing an error tuple which provides the detailed information about the failure (e.g. invalid WSDL address or connection time-out). This error message will be packaged into a reply tuple and then sent to the CCS. The reception of such an error reply will activate a number of adaptation rules (ARs) defined in the solution of CCS to execute the relative countermeasures. By selecting another chemical service (noted as CWS_i') that can provide the same functionality and then updating the binding reference of the failed task, a new invocation message is generated for CWS_i' to re-execute the failed task. In the following, we will use the "Best Garage" as a proof-of-concept example to illustrate the implementation of the binding-level adaptation in terms of chemical reactions.

Adaptation scenario 1. The invocation to the *WS-payByCard* Web service may fail. The cause can be manifolds, for example, a temporary network problem may lead to *WS-payByCard* inaccessible. In this case, the task t_4 will be automatically rebound to $CWS - payByCardPro$, which provides the same functionality as *CWS-payByCard* but promises the 100% availability at the cost of 0.1 euro per invocation.

In this scenario, once the payment using credit card is failed, *CWS-payByCard* chemical service will reply to *CCS-BestGarage* with an error message. Then, this error reply will be passed into the sub-solution of the corresponding SBA instance by CR *correlateResult* (as defined in Program 3.11). In the following, the presence of an error reply in the sub-solution of an SBA instance will trigger a series of chemical reactions in the middleware

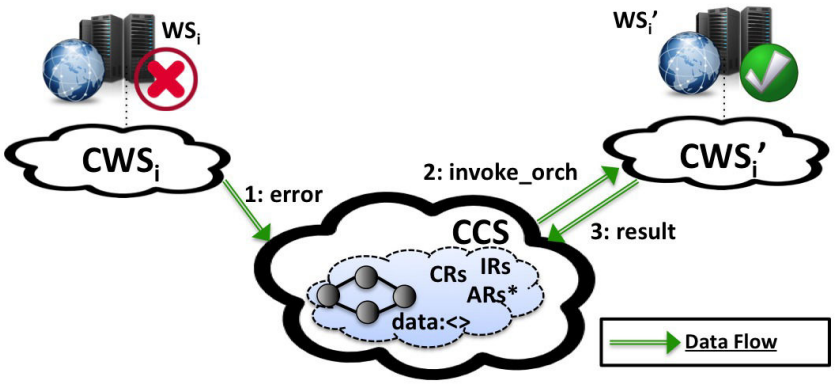


Figure 3.16: Service Orchestration Model: Binding-Level Adaptation

Program 3.20: Adaptation Rule for Binding-Level Adaptation (ARs)

```
let detectFailedTask =
  replace "Reply":cws:<"error":<?w1>>,
    "Task":ti:fi:"Invoking":<?w2>:binding
  by "Task":t_i:f_i:"Failed"<w2>:binding,
  if cws.equals(binding)
let adaptPayByCardFailure =
  replace "Task":"t4":"billing":"Failed":<?l>:"CWS-payByCard",
  by "Task":"t4":"billing":"Executing":<?l>:"CWS-payByCardPro"
```

to execute the corresponding binding-level adaptations that aim to resume the execution.

1. First, the emergence of such an error reply will activate the AR *detectFailedTask* in the sub-solution of this instance . As defined in Figure 3.20, it can detect a reply tuple with error information and then marks the runtime state of the corresponding task as "Failed". In the context of the *adaptation scenario 1*, the state of task t_4 is set to "Failed".

2. The change of the state of task t_4 will activate the AR *adaptPayByCardFailure*. As defined in Figure 3.20, it rebinds task t_4 to a new chemical service *CWS-PayByCardPro* and changes the state of t_4 back to "Executing". By this means, the state of the whole SBA instance rolls back to the moment before *CWS-payByCard* was previously invoked last time. The difference is that a different constituent chemical service is bound and invoked this time.

3. In the following, the IR *invokeT4* defined in Program 3.12 becomes active again to prepare an invocation request to re-execute task t_4. As we have introduced in Section 3.3.1, a new invocation request for invoking *CWS-PayByCardPro* activates the IR *prepareInvocationMsg* again to generate a "To_send" tuple with the concrete invocation message. By this means, the invocation message will be passed to the solution of *CWS-PayByCardPro* so that the execution of this SBA instance can complete with success.

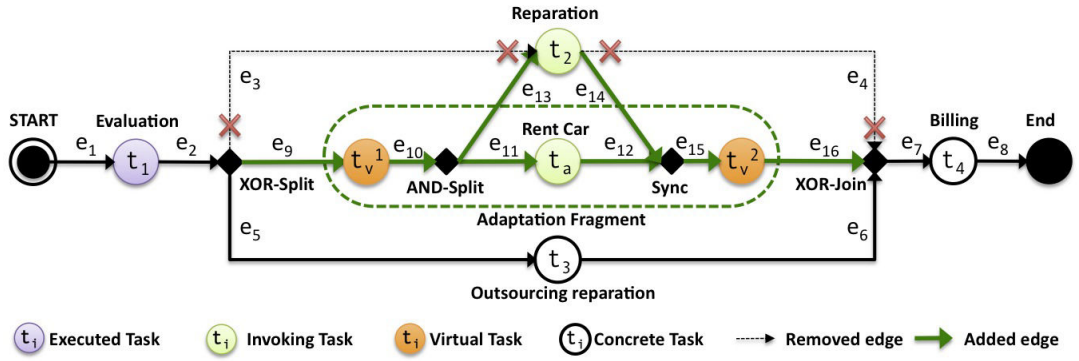


Figure 3.17: Scenario: Workflow-Level Adaptation of SBA

3.3.3.2 Workflow Level Adaptation of SBA

Sometimes, the SBA is required to be adapted on the workflow-level in order to achieve the ultimate business goal. The workflow-level adaptation refers to the modification of the workflow structure for a running SBA instance, namely to add or to remove tasks on the fly. Such workflow-level adaptation is similar to the workflow transformation process introduced in Section 3.2.2. We use the following scenario to present how adaptation can be performed on the workflow level.

Adaptation scenario 2. Since the problem of a car may be much more complicated than formerly estimated, the reparation may take longer time than expected. Remember that after the execution of task t_1, the expected time duration for the reparation (q_t) has been estimated by a technician, which can be seen as a part of the SLA between the client and the "Best Garage". In order to avoid the reputation degradation, the *"Best Garage"* expects to rent a car for its client until the reparation completes.

In order to implement this adaptation scenario, a new task t_a has to be added in parallel with the delayed task (either t_2 or t_3 depends on the execution context). Task t_a invokes *CWS-fastRentCar* chemical service which provides car rental service. We assume that the delay is caused by the execution of task t_2. As a result, the *adaptation workflow fragment* will be connected in parallel with task t_2, as depicted in Figure 3.17. First of all, since t_2 and t_a are executed in parallel, an AND-split and a synchronization workflow patterns are used to connect both tasks. Then, in our workflow model, two complex workflow patterns cannot be directly connected. As a result, two virtual tasks t_v^1 and t_v^2 are required to be added in order to perform the coordination task. Both virtual tasks connect the adaptation fragment to the original workflow. Finally, task t_2 updates its neighbor information by disconnecting the links to both task t_1 and t_4. In this case, both edge t_3 and e_4 in Figure 3.17 will be removed.

In the context of service orchestration, the CCS implements the centralized control on the execution of workflow. The sub-solution of each running SBA instance includes a number of *monitor rules* which monitor the progress of each running SBA instance. If a delay is produced during the reparation process, the runtime state of the relative executing task is changed from "Invoking" to "Delayed". Due to the limited space, the definitions of monitor rules are not provided here. Using the scenario presented in Figure 3.17 as an example, we assume that the state of task t_2 is marked as delayed. The change of its runtime state will lead to a series of chemical reactions that perform the above-introduced workflow transformation process, controlled by a number of adaptation rules (ARs) defined

Program 3.21: Adaptation Rules for Workflow-Level Adaptation

```
01 let adaptWorkflowDelay =
02   replace "Task":t_i:f_i:"Delayed":<?w>:cws
03   by  "Task":"t_v1":Null:"Executed":<"Out":"AND-Split":<t_i, "t_a">>:Null,
04     "Task":"t_v2":Null:"Concrete":<"In":"Sync":<t_i, "t_a">>:Null,
05     "Task":t_i:f_i:"Invoking":<"In":"Seq":<"tv1">,"Out":"Seq":<"t_v2">>:cws,
06     "Task":"t_a":"carRenting":"Executing":<
07     "In":"Seq":<"tv1">,"Out":"Seq":<"t_v2">>:"CWS-fastRentCar"
08     connectAdaptFragToT1, connectAdaptFragToT4, invokeTa
09   if t_i.equals("t2") || t_i.equals("t3")
10 let connectAdaptFragToT1 =
11   replace-one "Task":"t_v1":Null:"Executed":<"Out":wfp:<t_d,"t_a">>:Null,
12     "Task":"t1":f_1:"Executed":<"Out":"XOR-Split":<t_out,?p>,?l>:cws
13   by  "Task":"t1":f_1:"Executed":<"Out":"XOR-Split":<"t_v1",p>,l>:cws
14     "Task":"t_v1":Null:"Executed":<"In":"Seq":<"t1">,"Out":wfp:<t_d,"t_a">>:
           Null
15   if t_d.equals(t_out)
16 let connectAdaptFragToT4 =
17   replace-one "Task":"t_v2":Null:stat1:<"In":wfp:<t_d,"t_a">>:Null,
18     "Task":"t4":f_4:stat2:<"In":wfp:<t_in,?p>,?l>:cws
19   by  "Task":"t4":f_4:stat2:<"In":wfp:<"t_v1",p>,l>:cws
20     "Task":"t_v2":Null:stat1:<"Out":"Seq":<"t4">,"In":wfp:<t_d,"t_a">>:Null
21   if t_d.equals(t_in)
```

in Program 3.21.

1. Firstly, the AR *adaptaWorkflowDelay* is able to detect a delay during the reparation and it resets the state of the delayed task back to (task t_2 in the above-mentioned scenario) "Invoking" . Moreover, it creates a number of new tasks that define the *adaptation fragment*. 1) task t_{v1}: it defines the virtual task t_v^1 with a runtime state "Executed" (L.03). It has two successors executed in parallel: the delayed task t_i and the new task t_a. 2) task t_{v2}: it defines the virtual task t_v^2 which is not executed yet (L.04). It synchronizes the inputs from both the delayed task t_i and the new task t_a and it has only one successor t_4. 3) task t_a: it is bound to *CWS-fastRentCar*, which provides the car rental service (L.06-07). It has only one precedent t_{v1} and one successor t_{v2}. The runtime state of t_a is set to "Executing", which will activate the IR *invokeTa* in succession to start the execution of this task.

2. Meanwhile, four rules have been added. 1) *connectAdaptFragToT1*: it connects the adaptation fragment to the task t_1. The connection is performed by modifying the neighborhood information of the related tasks. As defined in Program 3.21, it reads the successors of task t_{v1} (L.11), one of the successor is fixed (t_a) and the other (t_d) represents the delayed task. And then, it removes the edge between t_1 and t_d, and builds a new edge between t_1 and t_{v1}. 2) The rule *connectAdaptFragToT1* is similarly defined to connect the adaptation fragment to task t_4. 3) *invokeTa*: as defined in Program 3.22, it prepares and sends an invocation message to *CWS-fastRentCar* in order to execute task t_a. 4) *invokeVirtualTask*: as we will explained later, it dictates the execution of a virtual task.

3. After the execution of the rule *adaptaWorkflowDelay*, all four new rules become active. Thus, a series of chemical reactions will take place concurrently, as described above. After these reactions, the instance solution becomes inert, and both task t_2

Program 3.22: New Invocation Rule for Executing the Adaptation Fragment

```
let invokeTa =
  replace "Task":"t_a":"carRenting":"Executing":<?w>:cws
  by  "Task":"t_a":"carRenting":"Invoking":<w>:cws,
    "Invoking":"CCS-BestGarage":<"operation":"rent">
let invokeVirtualTask =
  replace "Task":t_v:Null:"Executing":<?w>:Null
  by "Task":t_v:Null:"Executed":<w>:Null
```

and t_a are being executed in parallel. Then, after the reparation, the requester is notified to come to get its repaired car. The completion of the execution of both t_2 and t_a active the CRs presented in Figure 3.18 to execute synchronization workflow pattern. In the context of this new workflow, the execution of workflow arrives to the virtual task t_{v2}. As a normal task, its state is changed to "Executing".

4. The execution of a virtual task is performed by the rule *invokeVirtualTask* defined in Program 3.22. As we have introduced, the virtual task has no real functional requirement, it is only a place holder. Thus, once its state is marked as "Executing", the rule *invokeVirtualTask* will reset its state as "Executed". In the following, the CR *sequence* presented in Program 3.14 will be activated to start the execution of task t_4. The rest of the execution is similar to the process that we have presented in Section 3.3.1.

3.4 Decentralized Models for Adaptive Execution of SBA

As we have analyzed in Section 1.3, the orchestration model presents some performance limitations in scalability, throughput and execution time due to the centralized point of coordination. In this section, we are going to introduce two decentralized models for executing service compositions in the middleware: namely *semi-choreography* and *auto-choreography*. The former relies on the distribution of workflow fragments and coordination rules to realize decentralized service coordination; whereas the latter regards the service composition as an autonomic cell, which can be cloned, fused, and passed among the solutions of all the constituent services for the execution.

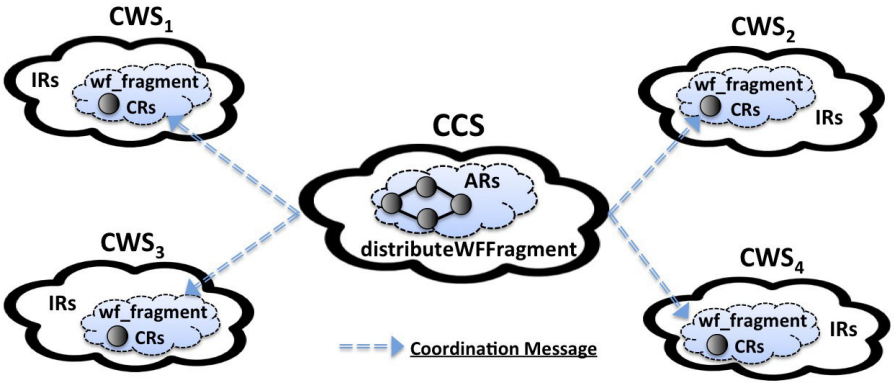
3.4.1 Semi-Choreography Model

The semi-choreography model separates the control of coordination from the control of adaptation. Each constituent service is able to coordinate the interaction messages with other participants, but it cannot react to runtime failures. All the failures have to be processed by a centralized adaptation engine, namely the solution of CCS[6].

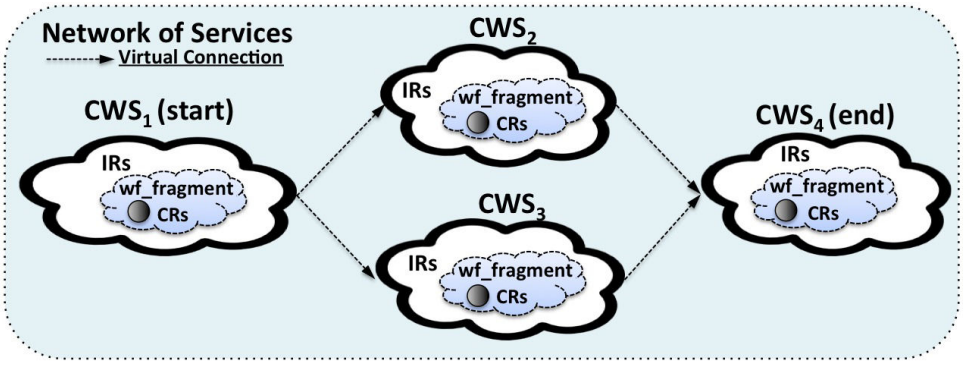
3.4.1.1 Execution of Workflow: Decentralized Coordination of Services

In the context of service choreography, each constituent service can directly interact with other participants (constituent services) without a centralized point of control (e.g. a

[6]That is why the word "semi" is used to define this model: from the perspective of service coordination, it implements the service choreography model; however, from the perspective of adaptation, it is still similar to the orchestration model.



(a) Distribution of Coordination Information



(b) Network of Services

Figure 3.18: Semi-Choreography Model: Configuration of Network of Services

business process). Accordingly, service coordination is performed by all the participants in a collaborative way: each constituent service can be seen as an intelligent agent that is able to decide what to do according to the message exchanges with other participants. The semi-choreography model describes such decentralized service collaboration as a *network of services*[7], where all the constituent services can be seen as virtually connected as a network according to the global execution plan (data flow and control flow). A network of services models a service composition as a pipeline transport network system: the execution of a service composition is modeled as the transportation of goods (e.g. gases or liquids) through a (number of) pipe(s), from the source (task) to the destination (task).

Since all constituent services are developed and managed independently by different organizations, the key to implement decentralized coordination lies in establishing the network of services. In this context, the execution of a service composition is performed by two steps. First, the *configuration* step is to set up the network of services by means of distributing coordination information. Then in the *execution* step, a service composition is executed by a series of direct interactions between constituent services that follow the pre-configured network of services.

[7]Although the term *network of service* is used in other papers with different meaning, in this dissertation, a network of services refers to a virtually configured service choreography.

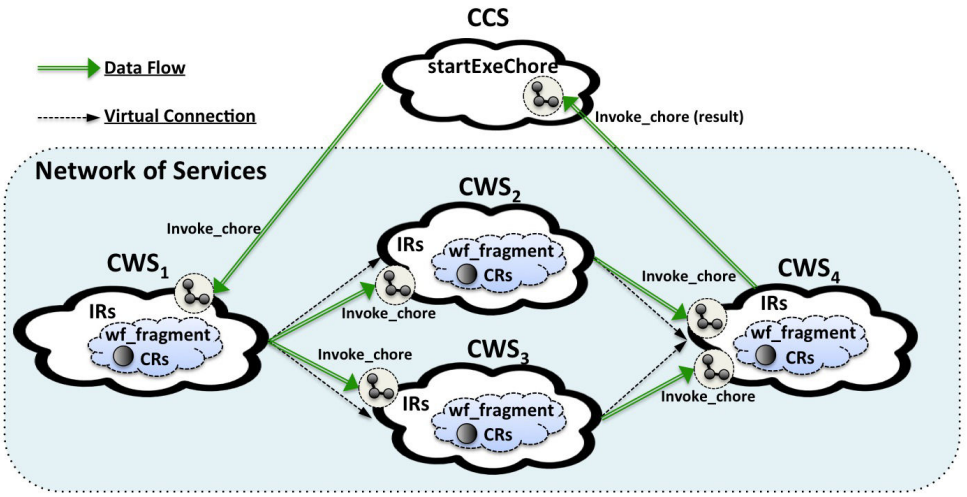


Figure 3.19: Semi-Choreography Model: Decentralized Execution of Workflow

Configuration: distribution of coordination information. In order to construct a network of services, each constituent service requires the coordination information in order to know: 1) the data flow information. For example, who are its neighbors, in other words, from whom it is expected to get the input(s) and to whom it is expected to forward to results; 2) the control flow information. For example, how to coordinate the incoming/outgoing messages with its neighbors, often expressed by the complex workflow patterns. As introduced, the concrete workflow describes both the data and control flow from a global viewpoint. Accordingly, in order to configure a service choreography, the CCS has to distribute (part of) concrete workflow description to all constituent services.

In this context, a number of rules are defined by the CCS to partition the concrete workflow into several pieces of *workflow fragment*s. Each fragment represents a part of global data flow and control flow from the local perspective of a specific participant. A workflow fragment is defined by a "WF_Fragment" tuple, described as follows:

"WF_Fragment":*signature_ccs*:*inst_id*:<task tuple(s), coordination rules>

It is composed of four parts: 1) a constant string ("WF_Fragment") as the declaration of a tuple for a workflow fragment. 2) The signature of CCS (*signature_ccs*). This is used by each constituent service to report a runtime failure, as introduced later on. 3) The instance identification information (*inst_id*). It is used for differentiating the workflow fragments from different instances of the same CCS. 4) a sub-solution which includes a (set of) task tuple(s) that present(s) a partial view on the global data flow and a number of reaction rules that describe the control flow.

In the following, CCS has to distribute these workflow fragments to all the constituent chemical services, as shown in Figure 3.18(a). After the distribution of workflow fragments, each constituent chemical service knows its neighbors and how to coordinate the interaction(s) with them. Accordingly, all the constituent chemical services can be seen as virtually inter-connected, and the network of services is constructed (described in Figure 3.18(b)). In Appendix A.1, we use the "Best Garage" as a proof-of-concept example to present the implementation of the semi-choreography model in the middleware.

Decentralized Execution Of Service Compositions. Having the network of services preconfigured, the execution of a service composition can start by sending an invocation

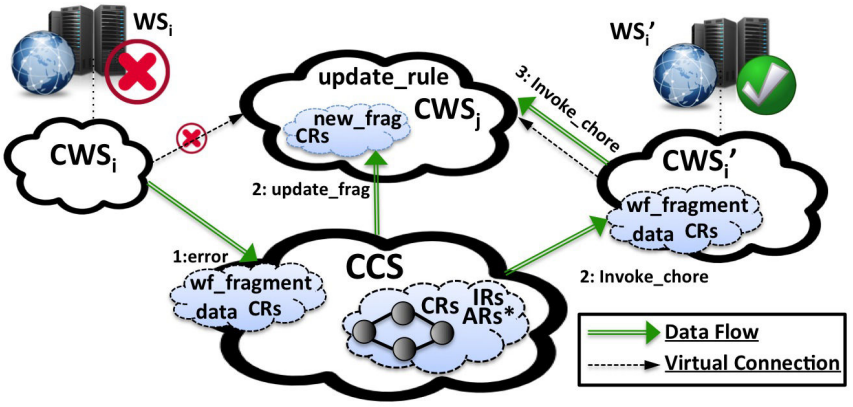


Figure 3.20: Semi-Choreography Model: Binding-Level Adaptation

message to the first CWS of the network of services, such as CWS_1 in Figure 3.19. An invocation message is defined by an "Invoke_chore" tuple, which is similar to an invocation message of the orchestration model, noted as:

$$\text{"Invoke\_chore"}:\mathit{from}:\mathit{inst\_id}:<?\mathit{parameters}>$$

An invocation message includes the information about the signature of the requester (*from*), the instance identification (*inst_id*), and a sub-solution of the invocation parameters. Once such an invocation tuple arrives to the solution of a constituent service, it will be moved to the sub-solution of the workflow fragment with the same instance identification. Then, the coordination rules defined in the workflow fragment are able to decide whether to start the invocation to the connected Web service or to wait for other invocation messages (e.g. for a synchronization task). Later, after the invocation to the concrete Web service, the computational result is written into the sub-solution of the corresponding workflow fragment. In the following, the coordination rules will generate and send a (number of) new invocation message(s) to (all the) succeeding CWS(es). In this context, the interactions are performed directly among all chemical services without the intervention of CCS, as illustrated in Figure 3.19. After the execution, the final result will be finally returned to the solution of the CCS. In Appendix A.2, we use the "Best Garage" example to demonstrate the implementation of such decentralized coordination of services.

3.4.1.2 Centralized Adaptation of Services

In the context of service semi-choreography, CCS distributes local coordination information (workflow fragment) to all the constituent services. Hence, the coordination of services can be performed in a collaborative way. However, the CCS does not tell constituent CWSes how to react to failures. This is because the execution of an adaptation plan sometime requires a global view on the service composition (e.g. to modify the workflow structure). Therefore, an adaptation plan cannot be successfully executed by a CWS if it has incomplete information. In this context, the adaptation of semi-choreography is performed in a centralized way. CCS acts as the centralized adaptation engine. Once a failure arises, it has to be reported to the CCS. By defining a number of adaptation rules, the CCS will then tell the corresponding CWS(es) how to copy with such failures.

In the following, we are going to illustrate the binding-level adaptation for the semi-choreography model. As depicted in Figure 3.20, once a failure arises during the invocation to Web service WS_i, the corresponding chemical Web service CWS_i will generate and send

an error reply to the CCS. Such an error response is described by an "Error_Invoke" tuple that encapsulates the entire workflow fragment for CWS_i, defined as follows:

$$\text{"Error\_Invoke"}:\text{ccs:inst\_id:}<?wf\_fragment>$$

The arrival of an error tuple in the solution of the CCS will activate the corresponding adaptation rules (ARs) which lead to a series of reactions to recover from a binding-level failure. Compared to the orchestration model, the adaptation of semi-choreography model exhibits a higher degree of complexity . Firstly, the CCS will update the molecular representation of workflow by replacing CWS_i by an alternative constituent service CWS'_i. Then, a new workflow fragment is generated and sent to CWS'_i with the expected invocation parameters. Meanwhile, the substitution of a constituent service can lead to the inconsistency problem for a network of services. As an example,CWS_j may be still waiting the input from CWS_i, although CWS_i is not involved in the service choreography anymore. Therefore, the CCS also has to send an updated workflow fragment for each of the succeeding services of CWS_i (e.g., CWS_j in Figure 3.20). In the following, each succeeding service will update its local copy of workflow fragment. By this means, the network of services is reconfigured.

On the other side, the arrival of a new workflow fragment to the solution of CWS'_i will lead to the execution of the failed task by invoking the Web service WS'_i, as illustrated in Figure 3.20. After the invocation, the coordination rules defined in the sub-solution of the workflow fragment will generate a new invocation message for CWS_j. By this means, the execution of workflow has successfully arrived to CWS_j, the rest of execution will be performed as formerly expected.

The workflow-level adaptation is performed in the similar way. The modification of workflow structure can also lead to the changes in the pre-configured network of services. Therefore, the CCS has to distribute an updated/new/cancelation workflow fragment for each constituent CWS that is involved in this modification. For example, due to the modification of the workflow, some unexecuted task(s) may be removed form the workflow. In this case, a cancelation message has to be sent to each corresponding CWS(es) since they will never be invoked any more. Since the implementation of adaptation for the semi-choreography model is complicated, with so many complex reaction rules defined, we do not provide the details of those rules in this dissertation for the reason of simplicity. The source code can be found at: `https://scm.gforge.inria.fr/svn/myriads/software/chen/2013/middleware/`.

3.4.2 Auto-Choreography Model

The semi-choreography model presents a high degree of complexity at both design time and runtime. On one side, its implementation requires a larger number of reaction rules which describe a series of more complex reactions than the orchestration model. On the other side, it is costly in establishing network of services as well as in adapting to runtime failures, since a great number of coordination messages have to be generated and distributed over the network. In this section, we are going to introduce *auto-choreography* model, which exhibits a lower complexity as well as a higher autonomicity and efficiency.

3.4.2.1 Decentralized Coordination of Services

Metaphor: composition cell. From the perspective of auto-choreography, a service composition is modeled as an autonomic *composition cell*, which encapsulates a set of

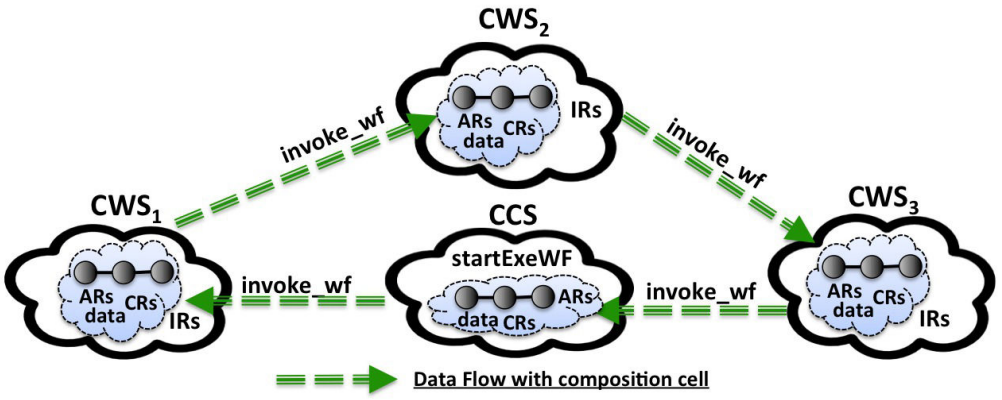


Figure 3.21: Auto-Choreography Model: Execution of Workflow

molecules to express a self-managed and self-adaptable service composition. In this context, a service composition is executed by the moving the corresponding composition cell among all constituent chemical services. Receiving a composition cell, a constituent service can read both data and coordination information from it, then do the related job, write new information (result) and finally forward it to the succeeding service(s). When the execution completes, the whole cell, including the final results, will be returned to CCS.

As a result, auto-choreography can be seen as a "moving" service orchestration. All the constituent services can act as the coordinator in turn. When a composition cell arrives to a CWS, all the reaction rules (CRs, IRs and ARs) are available by reading the contents from the composition cell. Thus each constituent service has the knowledge about the entire data flow (expressed by molecular workflow representation), control flow (expressed by CRs) as well as a collection of predefined adaptation plans (expressed by ARs). In this case, service coordination and adaptation (if necessary) can be thus performed locally in its solution.

As an example, Figure 3.21 illustrates the execution of a service composition by the auto-choreography model. Firstly, the CCS starts the execution by sending the composition cell to the first chemical service CWS_1. Then, the execution continues in the solution of CWS_1. The IRs in the composition cell can be activated to prepare the invocation parameters for executing the first task. CWS_1 is able to read these parameters from the composition cell and then invoke the corresponding concrete Web service. After the invocation, the result will be written back to the composition cell. In the following, the CRs in the composition cell can be activated to decide which task(s) to execute next. By reading such information[8] from the composition cell, CWS_1 will forward the composition cell to the succeeding participant, i.e., CWS_2. Then, the execution of the composition cell is carried out in the similar way in the solution of CWS_2 and CWS_3. Finally, the entire composition cell, including the final results, will be returned to CCS. In Appendix B.1, we will demonstrate the implementation of the auto-choreography model by using the "Best Garage" example.

3.4.2.2 Decentralized Adaptation of Services

The composition cell also includes a number of adaptation rules (ARs). Therefore, it is capable of reacting to runtime failures without requesting CCS. Once a failure happens,

[8]Actually, the information refers to the output molecules of a series of chemical reactions directed by a number of coordination rules in the composition cell.

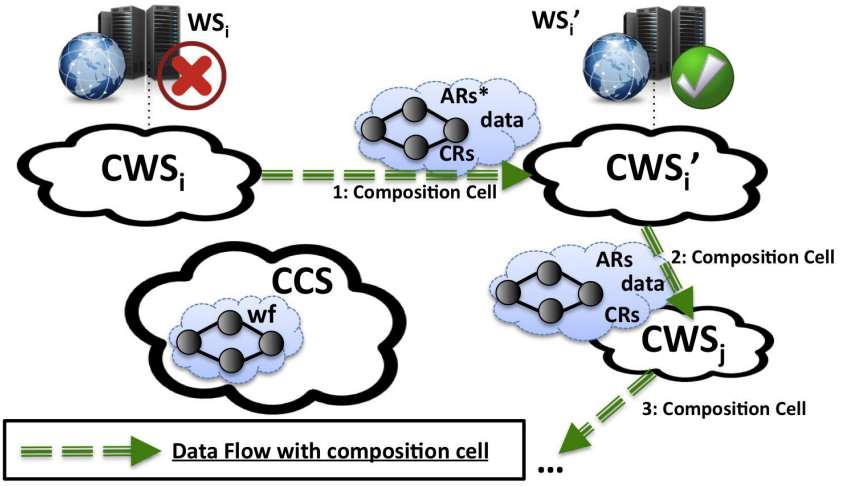


Figure 3.22: Auto-Choreography Model: Binding-Level Adaptation

the adaptation plan can be executed locally in the solution of a constituent service. The execution of adaptation plan is controlled by the same collection of rules as the orchestration model. In the following, we use the *adaptation scenario 1* (binding-level adaptation) presented in Section 3.3.3.1 as a proof of concept to illustrate how decentralized service adaptation is performed for auto-choreography model.

Figure 3.22 depicts the binding-level adaptation for auto-choreography. Once a Web service WS_i is failed, the corresponding chemical Web service CWS_i will receive an "Error" reply tuple which indicate the detail of failure (this is similar to the orchestration model). Such an error reply will inject into the corresponding composition cell and immediately a series of adaptation rules will be activated to change the binding information. Once the adaptation is performed, CWS_i is able to read the information about the new service composition from the composition cell and then it will pass the entire cell to the new chemical service, noted by CWS_i', which is expected to re-execute this failed task. CWS_i' connects to the Web service WS_i' which is able to provide the same functionality as WS_i. After the invocation, CWS_i' will forward the entire composition cell to the succeeding chemical service, for example, CWS_j. In this scenario, the adaptation is performed in a decentralized and autonomous way: each running CWS acts as the coordinator, the recovery can be performed locally and the execution is resumed autonomously.

The workflow-level adaptation can be performed in the same way. A number of ARs can be activated as runtime (the condition of activation depends on the specific adaptation plan) and then modify the workflow structure in the local solution of a constituent service. In the following, the execution of a composition cell (movement of the composition cell) will follow the new workflow. In Appendix B.2, we use the "Best Garage" as an illustrative example to present the implementation of both binding-level and workflow-level adaptation for the auto-choreography model.

Chapter 4

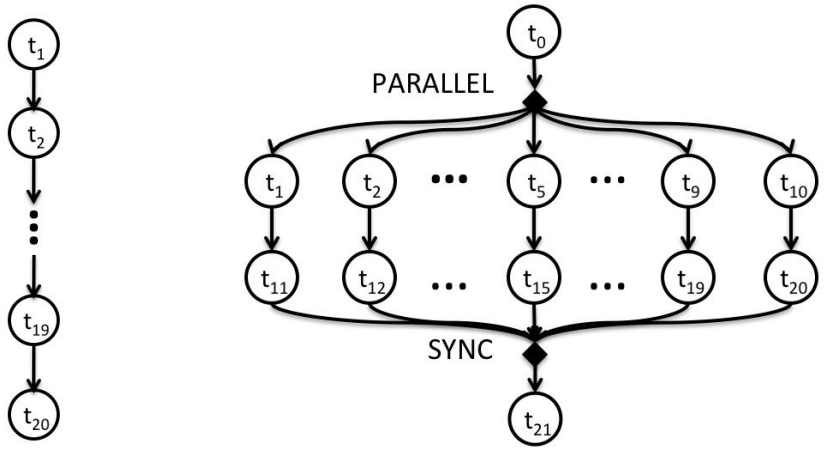
Evaluation: Implementation and Experimental Results

Abstract. In this chapter, the performance of the chemistry-inspired middleware is evaluated. The evaluation aims to compare the performance of the three coordination models presented in the previous chapter so as to find out the advantages as well as the limitations of each model under different execution contexts. Therefore, the evaluation results presented in this chapter is the basis for our future work in building context-aware service middleware, where an SBA instance can be executed by selecting the most suitable coordination model according to its runtime execution context. Section 4.1 first analyzes the performance of different coordination models in executing the experimental workflows by using a variety of performance metrics. Then, the distributed implementation of the middleware is presented in Section 4.2. Finally, in Section 4.3, a number of experiments are conducted by executing both experimental workflows using different coordination models in the middleware. The experimental results prove our analysis in Section 4.1.

4.1 Performance Analysis of Different Execution Models

The discussion in this chapter is based on two different types of *experimental workflow*s depicted in Figure 4.1.

- *LONG_SEQ* workflow presents the execution of a long sequence of tasks. The execution starts from task t_1 and terminates with task t_{20}. All the tasks are executed in sequential order.
- *WIDE_PAR* workflow presents the parallel executions of a large number of tasks. After task t_0, the execution is diverged into 10 parallel branches, and each branch includes 2 tasks. When the executions in all branches have been completed, task t_{21} will finally be executed.



(a) LONG_SEQ (b) WIDE_PAR

Figure 4.1: Experimental Workflows

In the following, we are going to analyze the performances of different coordination models by means of various performance metrics. The analysis result is summarized in Table 4.1 at the end of this section.

4.1.1 Complexity

Our discussion starts with the complexity of different models, from both design-time and runtime perspectives.

Design-Time Complexity

Generally speaking, the chemistry-inspired middleware presents a low complexity at design time. A chemical Web service (CWS) can be created automatically by using *CWSCreator*. It is a command-line tool provided by the middleware that is able to create a chemical Web service by requiring two parameters: 1) the expected name of the CWS and 2) the URL (if the WSDL file is available via Internet) or the path (if the WSDL file is accessible from the local disk of the computer) of the WSDL file that describes a real-world Web service. By parsing the WSDL file, an HOCL program can automatically be generated which defines a chemical solution with all the expected molecules (e.g. connector, etc.)

and rules (e.g. IRs, etc). Hence, if a Web service is already implemented outside the middleware (with an available WSDL), the related chemical image can be easily created in the middleware without any prerequisites on chemical computing.

The development of a CCS is more complex. Currently, an SBA provider is required to manually write an HOCL program that defines the chemical solution of CCS with all expected molecules and reaction rules. However, as we have demonstrated that most of BPEL constructs can be expressed by using generic reactions rules, one of our ongoing work is to develop a tool, named *CCSCreator*, that is able to generate the HOCL program for CCS by parsing a BPEL business process: the molecular representation of workflow and all the coordination rules for the service orchestration (since a BPEL process defines a service orchestration) can be automatically created. Nevertheless, since each SBA provider has different adaptation plans, (s)he has to define the adaptation rules by hand. In the future, we aim to develop some graphic tools to facilitate SBA providers to specify adaptation plans (specifying the change of bindings, etc.). By using these tools, the complexity in the development of a service orchestration can be greatly reduced.

From the previous chapters, we can see that most of coordination rules, invocation rules and adaptation rules for the orchestration model can be directly re-used by the auto-choreography model. Only few rules require some minor modifications. Therefore, once the reaction rules for orchestration model have been defined, the development of auto-choreography brings little additional design-time complexity. By contrast, compared to auto-choreography model, semi-choreography requires to re-write a respectively larger amount of reaction rules, which lead to a higher complexity at design time.

Runtime Complexity

Having analyzed the design-time complexity, we are going to discuss the runtime complexity of different models.

Coordination complexity. The execution of a service orchestration is simpler compared to both choreography models, since each constituent service only needs to interact with the centralized coordinator: it receives an invocation request, performs the computation and returns the computational results. A CWS does not need to know any details about the service composition (such as the coordination information). Compared to orchestration model, auto-choreography presents a higher coordination complexity. To execute a service composition, each constituent service has to interpret the composition cell in order to decide what to do. Additionally, to execute parallel branches, the cell cloning and cell fusion processes can bring additional runtime complexity. Semi-choreography is considered as the most complex one since it requires the partitioning of workflow and the distribution of workflow fragment in order to pre-configure the collaboration of services before the execution can actually start. During the execution, when an invocation message arrives, a CWS also has to interpret the corresponding workflow fragment in order to know what to do.

Adaptation complexity. Runtime adaptation of a service orchestration is simple due to the centralized adaptation engine. Once a failure arises, a CWS only needs to report the failure to CCS, and CCS is able to react to these failures by generating a new invocation message to another alternative chemical service. By contrast, semi-choreography presents the highest complexity in service adaptation. Similar to orchestration model, it also relies on a centralized adaptation engine. However, each modification of the service composition

(on either binding or workflow level) will result in the generation and the distribution of a number of updated workflow fragments. All related CWSes have to update their local copy of workflow fragment in order to guarantee the consistency of the network of services. Finally, compared to the centralized adaptation engine, auto-choreography has a medium complexity. On one side, since the adaptation can be executed locally by each constituent service, the complexity of adaptation is reduced. On the other side, if some global modifications (e.g. workflow-level adaptation) are required, the cell cloning and cell fusion introduce additional complexities.

4.1.2 Cost

The cost in executing a service composition is evaluated by 1) the number of messages interchanged between the CCS and CWSes; 2) the network traffic raised by passing these messages. In the following, we use both experimental workflows to investigate the cost of service coordination and adaptation.

Cost in Service Coordination

Number of messages. First, to execute the *LONG_SEQ* workflow by the orchestration model, the CCS needs to interact with each of the 20 constituent services only once and each interaction requires two messages, namely an invocation message and a response message. Therefore, the total number of exchanged messages is 40. Using semi-choreography model, in the configuration phase, the CCS has to firstly distribute a coordination message to each of all 20 constituent services (thus totally 20 messages). Then in the execution phase, the CCS first sends an invocation messages to service S_1; then 19 interactions are needed for executing the workflow; finally S_{20} will return the final result to the CCS (totally 21 messages). Thus, the total number of exchanged messages comes to 41 for semi-choreography model. By contrast, auto-choreography model does not require the configuration phase, the execution of workflow is similar to semi-choreography model. As a result, it requires only 21 messages to execute a service composition.

The execution of the *WIDE_PAR* workflow is performed in the similar way. The orchestration model requires 44 messages since two messages are transmitted to interact with each of the 22 constituent services. Semi-choreography model has to distribute 22 coordination messages to all the constituent services in the configuration phase. And then, it requires 32 messages to execute the service composition. Accordingly, the total number of messages comes to 54. Finally, auto-choreography model only needs 32 messages to get the final computational result.

From the above discussion, we can conclude that auto-choreography generates the fewest number of messages. Since semi-choreography relies on the distribution of coordination information, the greatest number of message exchanges are required. Finally, the orchestration model generates a medium number of messages compared to the other two models.

Network traffic. In order to evaluate the network traffic, we have the following assumptions. First we suppose that each constituent service replies what it has received. In this context, all the invocation messages have the same size, noted by T_{Inv}. Second, each workflow fragment (for the semi-choreography model) is supposed to have the same size, noted by T_{Frag}. Finally, the total size of all the workflow fragments is roughly considered to be equal to the size of the composition cell, noted as T_{Cc} ($\sum_{i=1}^{N} T_{Frag}(t_i) = T_{Cc}$, with N the total number of tasks).

In this context, to execute the *LONG_SEQ* workflow, the communication cost for orchestration model equals to 40*T_{Inv}. The network traffic of semi-choreography is composed of two parts: T_{Cc} (cost for distributing all workflow fragments) + 21*T_{Inv} (cost for coordinating constituent services). The communication cost of auto-choreography model equals to 21*T_{Cc}, since the entire composition cell is transmitted during each interaction between constituent service. The network traffic for executing the *WIDE_PAR* workflow can be evaluated in the similar way. The communication costs for orchestration model, semi-choreography model and auto-choreography model are respectively 44*T_{Inv}, T_{Cc} + 32*T_{Inv} and 32*T_{Cc}.

Cost in Service Adaptation

Runtime adaptation aims to modify service compositions by executing adaptation plans. As we have introduced, the modification can be performed either on the binding level or on the workflow level. Different SBA providers may define various adaptation plans. However, all binding-level adaptation plans behave in the similar way, namely to replace the failed service by an alternative one. Therefore, we are going to analyze the scenario of binding-level adaptation presented in Section 3.3.3.1 in order to investigate the cost of adaptation.

First of all, in the context of service orchestration, a binding-level adaptation requires 2 additional messages (an error message and a new invocation message) to execute a task compared to a regular execution, as illustrated in Figure 3.16. So the additional network traffic is 2*T_{Inv}[1]. The semi-choreography model is also based on a centralized adaptation engine but it is more costly than the orchestration model. As illustrated in Figure 3.20, apart from an error message and a new invocation message, the CCS also has to distribute a (number of) updated workflow fragments to the neighbor(s) of the failed task. Thus, the additional network traffic comes to 2*T_{Inv}+M*T_{Frag}, with M the number of neighbors of the failed task. By contrast, the binding-level adaptation for auto-choreography is simple, the failed CWS is able to execute the adaptation plan locally and then directly forwards the whole composition cell to the alternative CWS (refer to Figure 3.22). Thus, only one additional message is required, which lead to the additional network traffic by T_{Cc}.

From the above discussion, we can see that semi-choreography model presents a higher complexity in runtime adaptation compared to the other two models. The adaptation for both orchestration model and auto-choreography model generates few number of messages and little additional network traffic.

4.1.3 Efficiency

The efficiency of a coordination model is directly determined by its complexity and cost. In the following, we are going to investigate the efficiency of different models for service coordination and adaptation. The results of this investigation will be proved by the experimental results in Section 4.3.

End-to-End Execution Time.

The efficiency for service coordination can be directly measured by the end-to-end execution time of a service composition. To execute the same workflow, a more efficient model will result in less execution time. Firstly, we assume that each CWS takes the same time

[1]For the reason of simplicity, we assume that the size of an error message and the size of an invocation message are the same, namely equal to T_{Inv}

to response. In this context, the differences in the end-to-end execution time result from the transmission of messages. Then, we suppose that the network is fair and stable, which means the same speed is guaranteed for transmitting all the messages. Accordingly, the transmission time of a message is determined by the total size of all the messages.

We have analyzed the communication cost when we were investigating the network traffic. The total size of the interchanged messages for all three models have been calculated. As a result, when the size of each invocation messages is small (e.g. each constituent service only requires a string as the invocation parameter), T_{Inv} is negligible compared to T_{Cc} ($T_{Inv} \ll T_{Cc}$). In this context, the orchestration model leads to the least communication cost and auto-choreography model generates the greatest network traffic. By contrast, when a large amount of data exchange is required, the invocation parameter takes almost all the size of a composition cell ($T_{Cc} \approx T_{Inv}$). Due to the direct interactions between constituent services, the auto-choreography model results in the least network traffic but the orchestration model becomes the most costly one.

Reactive Time (to failures).

The efficiency for runtime service adaptation can be measured by the reactive time to failure. In the following, we use binding-level adaptation as an example to analyze the adaptation efficiency for different models. As we have analyzed before, the adaptation of service semi-choreography generates a number of updated workflow fragments. Therefore, its reactive time is respectively longer due to the distribution of these messages. By contrast, the orchestration model and auto-choreography model generate respectively 2 and 1 additional messages once a failure arises. Accordingly, they are less complex and more efficient. In case of big data exchange, the auto-choreography model is theoretically twice more effective than the orchestration model. However, when the size of invocation parameter is small, the reactive time of the orchestration model can be much faster than the auto-choreography model.

4.1.4 Flexibility

The flexibility refers to the ability to adapt to the changes in the conditions or circumstances on the fly. First, with a centralized adaptation engine, the orchestration and semi-choreography models present better flexibility for runtime service adaptation. An SBA provider is able to define and add new adaptation rules at runtime in response to the failures which are unexpected before the execution. However, this is impossible for the auto-choreography model. Because once the composition cell has left the solution of CCS, the CCS will totally loss the control over it. It does not even know where it is nor which constituent CWS is processing it. Therefore, all the adaptation plans have to be defined before the execution start. In this context, once an unknown failure arises, the composition cell will be suspended since it does not know how to react to such failures. An alternative solution is to return a "suspending" composition cell to the CCS in order to add new adaptation rules, which brings additional complexities.

Second, the orchestration model is considered more flexible than semi-choreography model due to the up-to-date runtime state of workflow execution. Since all the messages have to be passed by the CCS, the CCS has a clear view on the runtime execution state of each constituent service. However, in the context of semi-choreography, the knowledge of CCS on the global execution state of workflow may be inconsistent with the actual one. Therefore, the execution state of some tasks are unknown during the execution (CCS cannot know the execution state of a task except that a constituent service has reported

Table 4.1: Comparison of Different Models

Metrics	Orchestration	Semi-Chore	Auto-Chore
Complexity			
Design Time	Low	High	Medium
Runtime (Coordination)	Low	High	Medium
Runtime (Adaptation)	Low	High	Medium
Cost (Coordination)			
Num. of Msg	Medium	High	Low
Network Traffic (Large Data)	High	Medium	Low
Network Traffic (Complex WF)	Low	Medium	High
Cost (Adaptation)			
Num. of Msg	Medium	High	Low
Network Traffic (Large Data)	High	High	Low
Network Traffic (Complex WF)	Low	High	High
Efficiency			
Execution Time (Large data)	High	Medium	Low
Execution Time (Complex WF)	Low	Medium	High
Reactive Time (Large data)	Medium	High	Low
Reactive Time (Complex WF)	Low	High	Low
Flexibility	High	Medium	Low
Robustness	High	Low	Medium

a failure). In this case, some of the workflow-level adaptation plans cannot be executed since workflow-level adaptation usually can only operate on unexecuted tasks.

4.1.5 Robustness

The robustness refers to how well the execution of a service composition can cope with runtime failures. As we have demonstrated, all the three models are able to react to runtime failures raised from both functional or non-functional levels. However, they have different degrees of robustness.

Firstly, from the above discussions, we can see that semi-choreography presents the lowest degree of robustness. On one side, the execution of workflow-level adaptation plans may fail due to the inconsistent execution state. On the other side, its adaptation depends on the distribution of updated workflow fragments, which may become the potential cause of another failure. For example, an adaptation message may be delayed (the cause of a new non-functional failure) or even lost (the cause of a new functional failure). Then, compared to the orchestration model, auto-choreography is less robust because it may not successfully react to all kinds of failures (e.g. an unknown failure).

4.2 Implementation of Middleware

Each chemical service in the middleware, either a CWS or a CCS, is implemented by an HOCL program that defines a solution of molecules and reaction rules to realize the expected functionality. Most reaction rules defined by these HOCL programs are introduced in Chapter 3 and Appendix A-B. In this section, we provide the details in the implementation of the middleware. We first introduce the implementation of the HOCL compiler in Section 4.2.1 and Section 4.2.2. Then, Section 4.2.3 presents the distributed chemical

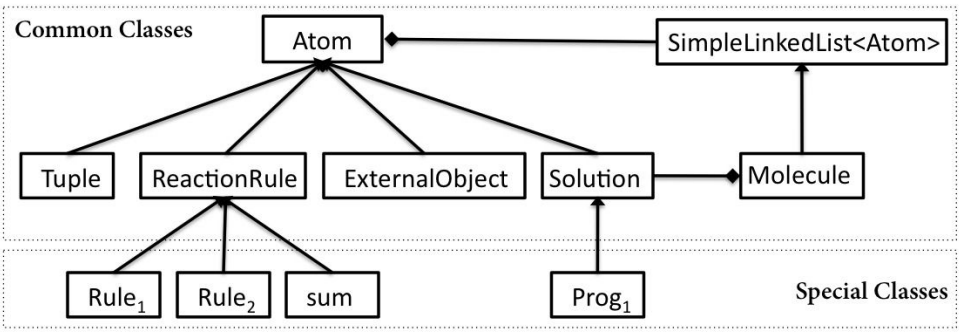


Figure 4.2: Java Implementation of Chemical Concepts

infrastructures on which the middleware can be running. Finally, the implementation of the "Best Garage" example based on the distributed chemical infrastructures is presented in Section 4.2.4.

4.2.1 Implementation of the HOCL Compiler

We will briefly introduce the implementation of the HOCL compiler. The HOCL Compiler is developed by *Dr. Yann Radenac* during his PhD thesis [124]. The compiler works on top of Java platform. As defined by the syntax of HOCL in Figure 2.4, first, all the chemical concepts can be seen as atoms. Thus, the HOCL compiler defines the *Atom* abstract class. An atom Java object can be constructed by one of the following types: *ExternalObject*, *ReactionRule*, *Solution* and *Tuple*, which describe respectively the basic values, reaction rules, chemical solutions and tuples. Second, a molecule can be an ensemble of one or more atoms. Therefore, the *Molecule* class implements a list to contain the atoms. Finally, a solution can contain any number of molecules. By this means, a solution object can contain a complex molecule that is composed of different types of atoms, such as tuples, reaction rules, basic values or even sub-solutions. The relationship of various Java class is depicted in Figure 4.2.

The execution of an HOCL program follows the process illustrated in Figure 4.3. First of all, the compilation of an HOCL program will generate a number of Java classes. Each reaction rule defined by an HOCL program leads to the generation of a new Java class by extending the *ReactionRule* class. The name of the rule is used as the name of the corresponding Java class. Moreover, since each HOCL program defines a chemical solution of molecules, an HOCL program is finally translated to a new Java class that extends the *Solution* class (e.g. the class *Prog1* in Figure 4.2). According, the compilation of an HOCL program will generate a set of new special Java classes (which extend the common ones).

Then, the execution of an HOCL program is thus performed by executing these Java programs. A new instance of the relative program class will be instantiated (e.g. new Prog1()). The *Solution* class defines a *reduce* method to carry out chemical reactions. For each rule defined in a solution, if reactable molecules are detected, they will by removed from the list, and new molecule(s) will be generated and added to the list. As an example, Figure 4.4 vividly illustrates a chemical program defined with several rules (namely $rule_1$, $rule_2$ and *sum*), integer numbers as well as strings. As a proof of concept, the definition of a specific rule *sum* is provided. It requires two integer numbers as the input molecules, and produces the sum of both integers as the output molecule. For each input molecule, an iterator is created. Then, both iterators have to traverse all the possible combinations[2] of

[2] As we analyze later, such brute-force methods for detecting reactable molecules causes the low efficiency

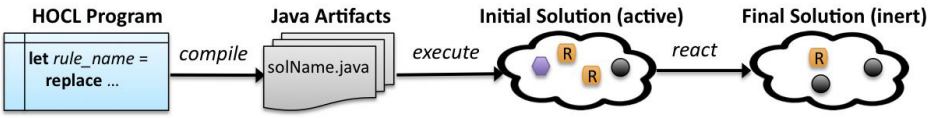


Figure 4.3: Execution of an HOCL Program

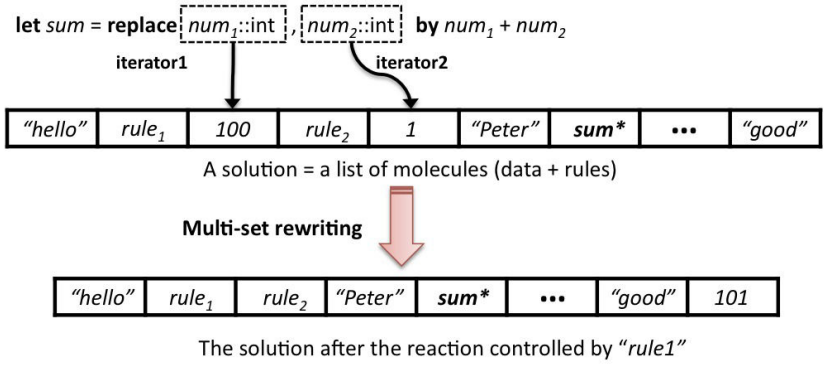


Figure 4.4: Implementation of Chemical Reactions

the molecules in the solution in order to discover possible reactions. As shown in Figure 4.4, once two integers are detected (e.g. 100 and 1), they will be removed and their sum (101) will be added to the end of the list. As a result, the execution of an HOCL program is performed by calling the *reduce* method of the program Java object. The execution continues until the solution arrives to the *inert* state, which means that all the remaining molecules in the solution are non-reactable according to all the chemical rules defined in the solution. And the computation results are finally left in the final solution.

4.2.2 I/O of HOCL programs

Using the original HOCL compiler, an HOCL program is modeled as a closed solution: all the elements have to be defined before execution. By this means, the reactions are performed as a one-shot process: when all reactable molecules have been consumed, the execution terminates. Hence, it cannot reflect the dynamic and evolving nature of chemical computing model. In response to such challenges, I have improved the compiler by introducing I/O mechanisms. A chemical solution is thus able to get the new elements either from the users or from remote HOCL programs.

Interaction with Users. In order to dynamically add or remove elements from a chemical solution at runtime, the *Solution* class also defines two additional methods.

- Firstly, the *add* method requires a *Molecule* object as input parameter, and adds it to the end of its internal list. When a molecule is added and if the state of this solution is marked as inert, the *reduce* method will be called since the new molecules may reactable with others according to some reaction rules.
- Furthermore, the *remove* method can get 1) a *Molecule* object and remove the first occurrence of this molecule from the list (if it is present) or 2) an index number and remove the molecule at the specified position in the list.

of HOCL.

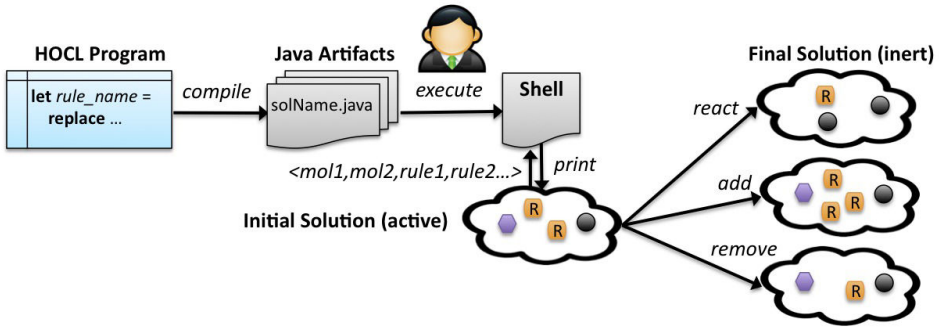


Figure 4.5: Interaction with Users

Then, each program class is generated with a shell program. When an HOCL program is executed, the reactions do not start immediately. Instead, the user enters to a shell, which provides a number of operations to manage the chemical solution of this HOCL program, as depicted in Figure 4.5.

put The *put* operation first enables the user to create a molecule of a specific type, and then adds the new molecule into the solution by calling the *add* method defined by the *Solution* class. Users can add 1) a new Java object by providing the parameters required by the constructor, as well as 2) a new chemical concept such as a tuple, a solution or even a rule.

get The *get* operation first lists all the molecules in a solution and enables the user to select a specific one to delete by calling the *remove* method defined by the *Solution* class.

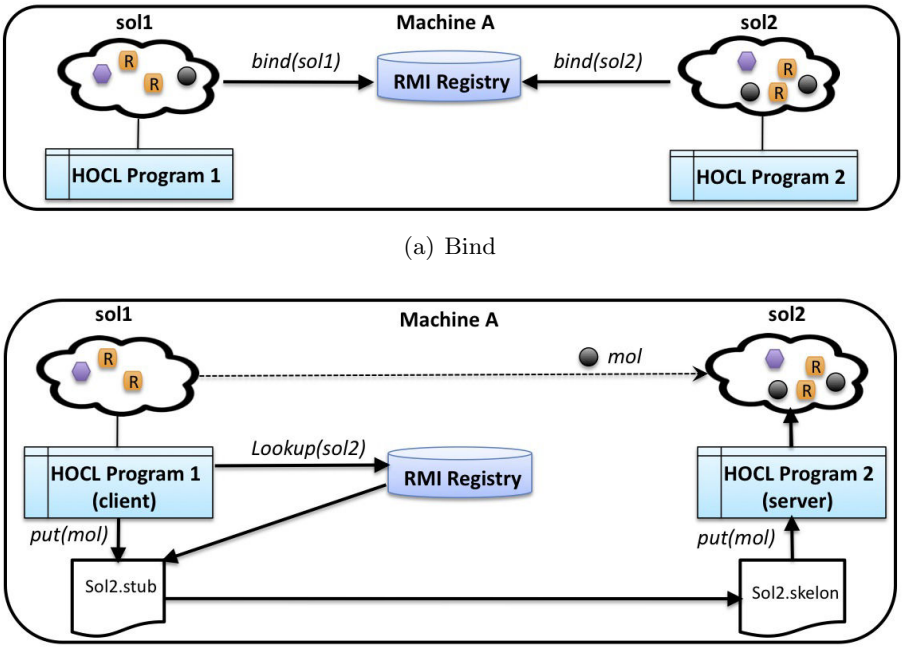
run The *run* operation starts the chemical reactions in the solution by calling the *reduce* method of the *Solution* class.

print The user can check the current content of solution by using the *print* operation, which calls the *toString* method of the *Solution* class in order to display the content of the solution on the terminal window.

By using these commands, users can manage the chemical solution at runtime more flexibly. More information about how to use the shell can be found in [143].

Interaction with Remote HOCL Programs. However, it is unpractical for the users to add new molecules by hand, especially for the case where a large number of data exchange is needed. Accordingly, it is more important to get molecules directly from other programs. By using the improved HOCL compiler, the HOCL program is translated into a number of Java artifacts which are running on top of Java RMI (Remote Method Invocation) framework. Some Java RMI modules are integrated for interactions between chemical programs.

First of all, each HOCL program registers its chemical solution with a unique name in the local RMI registry when it is executed. The registration is performed by calling the *bind* method defined by the Java RMI modules. As illustrated in Figure 4.6, two solutions of HOCL programs are registered respectively with the name sol_1 and sol_2. When an HOCL program (defined as the client) wants to interact with a remote one (defined as the server), it looks up the name of the server's program solution in the local RMI registry.



(a) Bind

(b) Lookup & Add Molecule

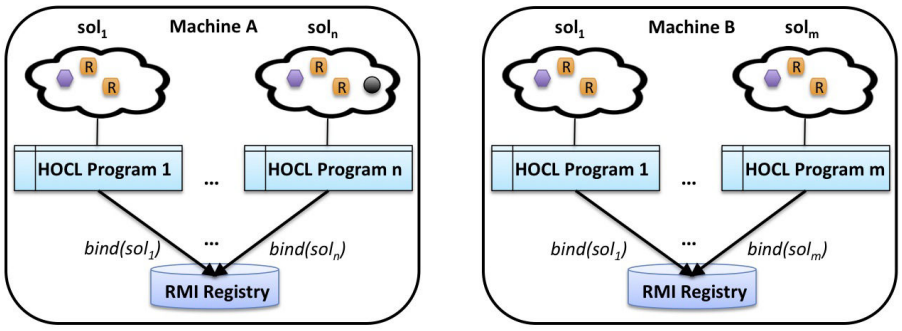
Figure 4.6: Interaction to a Remote Solution (Single Host)

And then, the client program downloads a *stub* of server's solution object, which has the same interfaces as the original one. By calling *put* method provided by the server's stub, the client is able to transfer any molecules to the stub. The stub marshals the arguments using object serialization, and sends the marshaled molecules to the server. On the server program side, the call is received by a *skeleton*, which is responsible for unmarshaling the arguments and invoking the server's implementation of the *put* method. By this means, an HOCL program is able to write any molecules to remote chemical solutions. This process is illustrated in Figure 4.6.

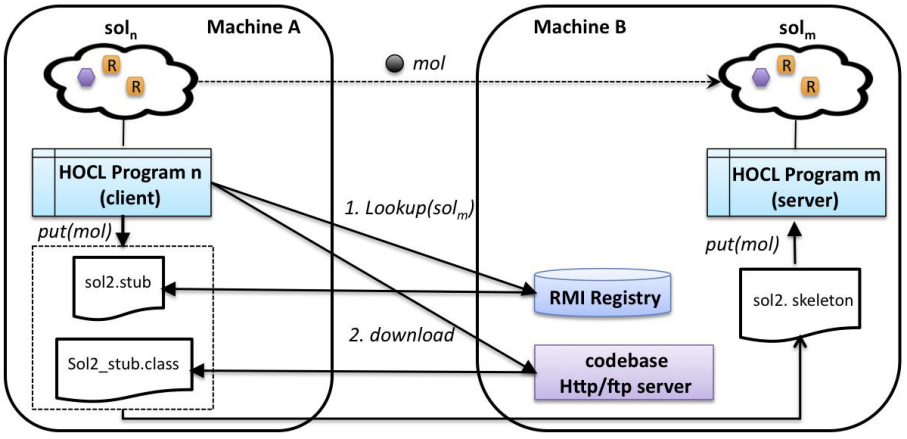
In the previous example, both client and server HOCL programs have to be running on the same machine. However, such remote interaction mechanism can be extended to the distributed implementations. Suppose that we have several machines, and on each machine, multiple HOCL programs are running. As illustrated in Figure 4.7, n HOCL programs are running on machine A and m HOCL programs are running on the machine B. First of all, every HOCL program registers with a unique name[3] in its local RMI register. Then, if the client program sol_k wants to write molecules to a remote solution sol_l which is located on a different machine, it has to look up sol_l in the registry of machine B, and gets the stub of sol_l. Since different JVMs are running on different machines, client HOCL program also needs to download the class definition of the sol_l from machine B. As a result, Machine B has to specify the code base pointing to a repository accessible publicly through Internet. Some protocols can be used such as HTTP server or FTP server. At this moment, the client program is able to interpret the stub of sol_l and then it performs in the similar way as the case that both programs are running on a single machine.

Automated Interaction with Reaction Rules By manipulating the RMI communicator, the *put* primitive is defined to encapsulate the entire process depicted in Figure 4.7

[3]Two solutions on different machines may have the same name.



(a) Bind



(b) Lookup & Add Molecule

Figure 4.7: Interaction to a Remote Solution (Multiple Hosts)

to write any molecules into a specific remote chemical solution. It requires two arguments: the signature of the server solution (*sol_name@dest_IP*) and the molecules to send (MOL). As introduced, the signature of the server solution is composed of two parts: 1) the name of the server solution (denoted *sol_name*) and 2) the IP address of the machine on which the server HOCL program is running (noted (*dest_IP*)). Therefore, by using the signature of the server solution, the client program is able to get the corresponding stub object as well as its definition. Then, the molecule MOL can be written to the remote solution by calling the *put* method defined by the stub. Therefore, several rules can be created to write a molecule of a specific type to a remote solution once the molecule is generated.

4.2.3 Distributed Chemical Infrastructure

Since an HOCL program is able to freely talk to remote HOCL programs running on distributed machines, the middleware can be running over the distributed infrastructures, defined as the *Distributed Chemical Infrastructures*. As illustrated in Figure 4.8, Firstly, each chemical service (either CWS or CCS) is defined by an HOCL program. Then, all HOCL programs are deployed over distributed infrastructures, such as a cluster or a federation of clusters, with a number of physical machines inter-connected by a Local Area Network (LAN) or the Internet. Each physical node is configured with an HOCL compiler for compiling and executing HOCL programs. As a result, each node can be seen as a Chemical Virtual Machine (CVM) which is capable to provide the chemical runtime

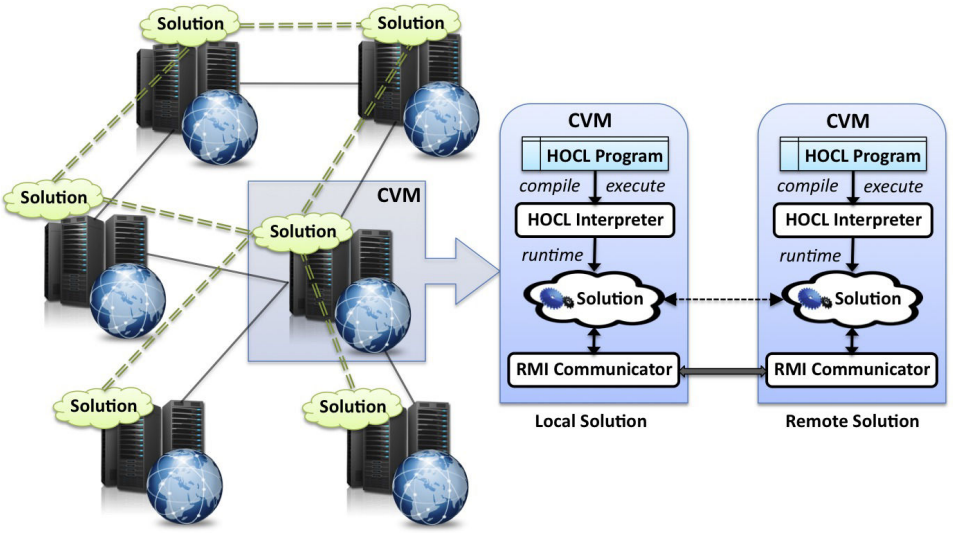


Figure 4.8: Distributed Chemical Infrastructures

environment for running chemical reactions. In our experiments, the distributed chemical infrastructure is set up over the Grid'5000 platform [2]. We have reserved 8 nodes in *paradent* cluster in Rennes, France, and a CVM is configured on each physic node.

4.2.4 Implementation of the "Best Garage" Example

As a preliminary prototype for validation, we have implemented the "Best Garage" illustrative example.

Implementation Setup. 1) First of all, an HOCL program is created as the chemical representation of SBA. It defines a solution which includes the molecular workflow representation as well as most of the above-introduced rules for service coordination, invocation and adaptation. 2) Each constituent Web service in Figure 2.7 is implemented as a JAVA Web service and deployed in Apache AXIS2 [16] Web service engine hosted on a single machine. In response to service invocations, each Web service sleeps for a certain time to simulate the execution time and then returns the result by randomly selecting an item from a predefined list. For example, *WS-SeniorT* predefines a table which provides a number of possible diagnostic results with different values for parameters (diagnosed problem, estimated time and cost). Once an invocation arrives, *WS-SeniorT* will randomly select an entry (represented by a line in this table) as the diagnostic result. 3) Using *CWSCreator*, an HOCL program is automatically created for each constituent service. 4) Finally, each HOCL program (1 CCS and 7 CWSes) is deployed on a distinct CVM.

Simulation of Different Scenarios. In order to validate our approach, we simulated all the scenarios that we have presented in this paper:

- Dynamic binding. We ran 100 SBA instances with cars of different types. AThus, task t_1 can be bound to either *CWS-SeniorT* or *CWS-JuniorT* according to the execution context.
- Binding-level adaptation. During all 100 executions, we temporarily removed the *WS-payByCard* Web service from the AXIS2 engine in order to simulate the in-

frastructure failure: the *WS-payByCard* Web service was not accessible within that duration. In this case, task t_4 is expected to rebind to *CWS-payByCardPro* service.

- Workflow-level adaptation. Both *WS-RepManager* and *WS-fastRepairCar* requires the estimated time (q_t) as one of input parameters. And their sleep time varies from 0.85*q_t to 1.15*q_t (uniform distribution). The solution of each SBA instance composes a timer, which monitors the execution time. If the estimated time (q_t) will be violated right away, it will put an *"ALERT"* molecule in the solution. This molecule will start the workflow-level adaptation.

Results. All 100 executions have been successfully completed: 1) for all 100 requests, task t_1 has bound to the correct diagnostic technician services; 2) 22 executions have encountered the failures while invoking to *WS-payByCard* Web service, and *WS-payByCardPro* service were successfully rebound; 3) 33 executions took longer time then expected; for each of them, the car rental service was bound and invoked.

4.3 Evaluation of Different Execution Models

In this section, we present a number of experiments conducted using the distributed implementation of chemistry-inspired middleware to evaluate the efficiency and complexity of different models that we have presented in Section 3.

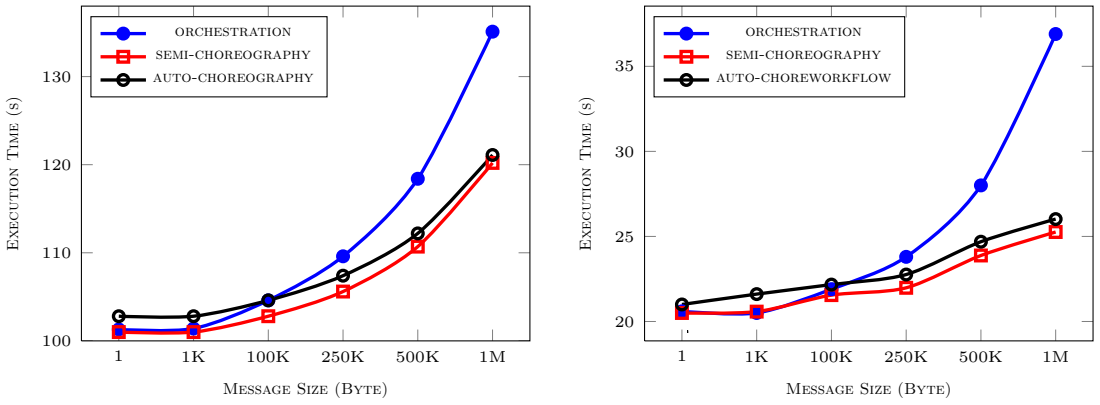
4.3.1 Experimental Setup

The experiments are based on the two experimental workflow presented in Figure 4.1. First, for each workflow, an HOCL program is developed to describe the corresponding CCS by defining its molecular presentation of workflow as well as a number of reaction rules (such as CRs, ARs and IRs). Each task t_i can be executed by two CWSes: the default chemical service cws_d_i and an alternative one cws_a_i. All the constituent chemical services are automatically generated based on two experimental concrete Web services.

In our experiments, we have implemented two experimental Web services in Java, noted respectively WS_1 and WS_2. WS_1 requires a string s as input, sleeps for 5 seconds and finally returns s as the result. WS_2 behaves in the same way as WS_1, except that it has a probability of τ_{cr} ($0 \leq \tau_{cr} \leq 1$) to return an error string message. An error message aims to simulate a runtime invocation failure. For example, the concrete Web service is not accessible due to network problem. Both Web services are running on Apache AXIS2 Web service engine [16] on a single machine outside the distributed chemical infrastructures.

For both CCSes, the default CWS of each task is created based on WS_2 whereas the alternative one connects to WS_1. In this case, the default CWS of each task cws_d_i has a possibility of τ_{cr} to reply with a an error tuple but the alternative one cws_a_i is always reliable. Each task t_i pre-binds to cws_d_i at the design time. Once a failure is reported by cws_d_i, the failed task t_i will rebind to cws_a_i. Since each constituent service returns what it has received, we can initiate an invocation message that is used to invoke all the constituent services. By this means, all the invocation messages have the same size, which satisfies our assumption in Section 4.1.2.

For each execution of an experimental workflow, all the chemical services (HOCL programs) are uniformly distributed to all the nodes of the distributed chemical infrastructures. By activating different sets of rules, the execution can start from CCS by different models. In the following, we present 2 groups of experiments to evaluate respectively the efficiency and complexity of different models.



(a) LONG_SEQ (b) WIDE_PAR

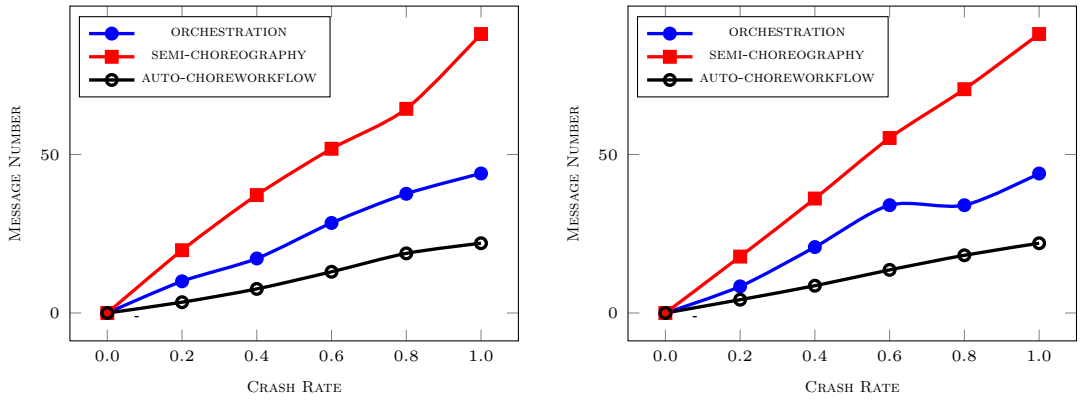
Figure 4.9: The Execution Time for Different Models

4.3.2 Experiment 1: Comparison of the Execution Efficiency

The first experiment is to evaluate the efficiency of different models by comparing the execution time. In this experiment, τ_{cr} is set to 0.0, which indicates that no error message will be generated and thus no adaptation is needed. As we have discussed, the size of each interaction message is determined by the size of the initial parameter. In order to simulate different execution contexts, the size of the initial parameter can vary from 0 (empty string) to 1MB (randomly generated string). For a given initial parameter, each workflow are executed 10 times for each model, and we calculate the average execution time. The ideal execution time (without consideration of transmission cost) of LONG_SEQ and WIDE_PAR workflow are respectively 100 seconds and 20 seconds. The result is presented in Figure 4.9.

Execution context 1: large workflow. When the size of invocation message is small, the communication cost for each invocation message is negligible. As a result, the orchestration model and the semi-choreography model have almost the same execution time. Both models perform efficiently in executing the experimental workflow: the coordination of more than 20 constituent services only brings less than one second extra time. Whereas, the auto-choreography model is less efficient in this context. As we have discussed in Section 4.1.3, although the auto-choreography model requires less interactions for an execution of workflow, if the size of a composition cell is much larger than the size of an invocation message ($T_{Inv} \ll T_{Cc}$), it still takes a little longer time to complete an execution of workflow since the transmission of a composition cell is much more costly than transmitting an invocation message.

Execution context 2: big data exchange. When the size of invocation parameter increases, the execution time of service orchestration increases more rapidly than the other two models. In the context where the big data exchange is required ($T_{Cc} \approx T_{Inv}$), the orchestration model requires more than 10% time to execute a workflow than both choreography models. This is because that the size of the workflow definition can be neglected compared to the size of the invocation parameter. Therefore, the size of a composition cell can be considered as the same to the size of the invocation parameter. But the orchestration model requires more numbers of interactions. Furthermore, the



(a) LONG_SEQ (b) WIDE_PAR

Figure 4.10: The Number of Additional Messages for Different Models

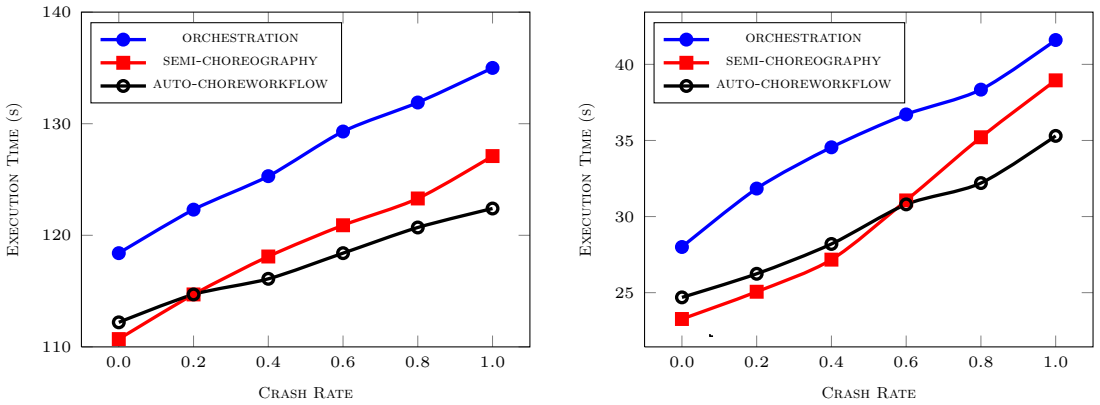
execution time of two choreography models become more close as the size of invocation parameter increases. This is because they follow the same interaction model and the size of the messages exchanged between constituent services for both models are almost the same (the semi-choreography model transmits the invocation parameter whereas the auto-choreography model transmits the composition cell).

4.3.3 Experiment 2: Comparison of the Adaptation Complexity

The second experiment is to investigate the adaptation complexity of different models. The crash rate of WS_1 τ_{cr} varies from 0.0 to 1.0 by a step of 0.2. For a specific τ_{cr}, each workflow are executed 10 times for each model, and we records the average overall execution time as well as the number of additional messages which are generated by runtime adaptation. In this experiment, the message size is set to 500KB.

Additional messages for binding-level adaptation. First, as Figure 4.10 demonstrates, the adaptation of auto-choreography is simple with only few additional messages generated. This is because of its autonomic nature, as we have analyzed in Section 4.1.3, the composition cell can be self-adapted locally by each constituent service and only one additional message is required for each binding-level failures. By contrast, the adaptation of the semi-choreography model depends on a centralized adaptation engine and the re-distribution of updated workflow fragments. Thus, it exhibits the highest complexity (in terms of a great number of additional messages). Compared to these two models, orchestration presents a medium complexity. Please note that in Figure 4.11(b), the experimental results in executing WIDE_PAR workflow by the orchestration model with a crash rate of respectively 60% and 80% have almost the same number of adaptation messages (represented by the blue line). This is accidental since two sets of experiments have encountered almost the same number of failures.

Reactive time for binding-level adaptation. Figure 4.11 illustrates the overall execution time of a workflow for different models when failures arise. As the crash rate increases, the execution time of both service orchestration and semi-choreography increase more rapidly than auto-choreography, for both experimental workflows. The gradient of



(a) LONG_SEQ (b) WIDE_PAR

Figure 4.11: The Overall Execution Time (Including Adaptation) for Different Models

each curve can reflect the adaptation cost, showing the additional execution time compared to non-failure execution, namely the total reactive time to failures.

Firstly, for the LONG_SEQ workflow, since all the tasks are sequentially executed, only one task can fail at one time. Both the orchestration and semi-choreography models rely on the centralized adaptation engine, they have the similar adaptation cost. Whereas for auto-choreography, the running CWS can execute adaptation actions locally. Accordingly it is the most efficient model which results in the best overall execution time when failures happen often.

For WIDE_PAR workflow, multiple failures can take place concurrently. In this case, auto-choreography still presents a high efficiency since each running CWS can execute adaptation actions independently (see Figure 4.11(b)). The other two models rely on a centralized execution engine but the performance of the semi-choreography model is a disaster when more and more failures arise. At this time, different from *LONG_SEQ* workflow, failures can take place concurrently and each failure will lead to the distribution of a number of updated workflow fragments. Accordingly, when failures occur often, it suffers from heavy transmission loads which leads to higher adaptation cost compared to the orchestration model.

4.3.4 Discussion

From the above presentation and experimental results, the following observations can be concluded.

- Orchestration model exhibits low design-time complexity and medium runtime complexity. It presents a low efficiency in executing the workflow with large data exchanges, but a low cost as well for executing runtime adaptation. Therefore, it is suitable for executing inter-organizational commercial workflow that requires less data exchanges and failures may arise often since all constituent services can be largely distributed over the world.

- Semi-choreography model has a high design-time complexity; but it can always result in better response time due to the direct interaction between constituent services. However, the runtime adaptation exhibits high complexity, especially in the context

where failures occur frequently. Accordingly, semi-choreography is suitable for building intra-organizational service compositions, where few failures can arise. Moreover, in this context, all workflow fragments can be pre-installed in all constituent services since the SBA manager has the control on all constituent services.

- Auto-choreography model brings medium design-time complexity. However, since the coordination and adaptation can be performed directly by each running service, it presents both low runtime complexity and high efficiency. Therefore, this model is suitable for executing data-intensive workflows, where a large amount of data exchange is required (e.g. scientific workflows).

Part III

Towards Proactive Adaptation of SBA

Chapter 5

A Two-Phase Online Prediction Approach

Abstract. In the previous chapters, we have presented how to flexibly react to runtime failures raised from either functional or non-functional level during the execution of SBA. However, such reactive adaptation adopts some *remedial actions* that can only ensure the successful completion of an execution of SBA rather than prevent the occurrence of these failures. The execution of remedial actions brings additional cost in terms of time consumption and other computational resources. Accordingly, the end-to-end QoS of SBA may degrade. As an example, the execution of an SBA instance may take respectively longer time when failures have occurred. By contrast, proactive adaptation aims to identify the needs for adaptation by forecasting an upcoming failure in the future. *Preventive action*s can be executed to improve the quality of the service-based systems before failures actually arise. As introduced in Section 1.4, both functional and non-functional failures can be avoided by applying proactive adaptations. In this chapter, our discussion will focus on the proactive adaptation for avoiding non-functional failures, for example, a running SBA instance fails to meet the expected end-to-end QoS. We are going to introduce a *two-phase online prediction* approach capable of drawing accurate and timely decisions for starting adaptation proactively in order to guarantee the end-to-end QoS of SBA.

5.1 Problem Statement: Challenges and Solutions

It is crucial for SBA providers to guarantee the end-to-end QoS of their SBAs since the end-to-end QoS directly determines the experiences of the end users. Better end-to-end QoS will enhance an end user's satisfaction as well as the reputation of an SBA. For instance, if an SBA always responds slowly, its clients may go to other SBA providers that can offer the same functionality.

5.1.1 Context: Prevention of Global SLA Violation

In this section, our discussion focuses on the context of preventing global SLA violation. As we have introduced in Section 1.1.4, some SBA providers may establish global SLAs with their end users. For example, an SBA provider promises to respond within a limited duration at a certain cost. The definition of the global SLA is based on all local SLAs between the SBA provider and each constituent service. The illustration of global and local SLAs can be found in Figure 1.5 and a concrete example was provided in Table 1.4. In this context, the degradation of the end-to-end QoS (e.g. delays) can result in the violation of global SLA, which will in succession lead to some undesirable results to SBA providers, such as reputation degradation as well as penalties. Accordingly, SBA providers have to guarantee the end-to-end QoS of each running SBA instance to avoid global SLA violation.

5.1.2 Challenges

Due to the loosely coupled and distributed execution environment, it is challenging for SBA providers to guarantee the end-to-end QoS. First of all, the SBA's end-to-end QoS is determined by the QoS of all constituent services. For example, the execution time of an SBA instance depends on how fast each constituent service responds. However, in such distributed execution environment, the QoS of each constituent service can be hardly ensured. For instance, a constituent service may take longer time to respond due to the network congestion. Additionally, since all constituent services are provided by different organizations, the SBA provider often lacks the control over these third-party constituent services, such as the management of their computational resources (e.g. infrastructures).

In this case, an SBA provider is required to monitor the incoming/outgoing messages of each running SBA instance in order to perceive some misbehaviors (e.g. the response of a constituent service may arrive later than expected). Accordingly, (s)he is able judge whether some preventive adaptation actions are needed by analyzing the runtime execution state of a running SBA instance. For example, if it notices that one of the constituent service responds late, a proactive adaptation plan can be executed to rebind (some) unexecuted task(s) to other alternative services with better expected response time. By this means, the end-to-end response time of this SBA instance can still meet the expected value defined in the global SLA.

Compared to reactive adaptation, proactive adaptation is more challenging. The *main challenge* is to accurately analyze the needs for adaptation at runtime in order to *accurately* trigger the preventive adaptation (represented by the *analyze* phase in MAPE control-feedback loop presented in Figure 1.10). In the context of reactive adaptation, adaptation decisions can be easily made since the need for adaptation is straight-forward: each failure will immediately trigger the execution of relative adaptation plans for recovering the execution. By contrast, for proactive adaptation, adaptation decisions rely on

the prediction of a failure[1] in the future. An accurate prediction approach has to meet the following requirements (challenges):

- *Effectiveness.* An effective approach can successfully predict as many SLA violations as possible. In the extreme cases, the most effective prediction approach can successfully predict all the global SLA violations.
- *Precision.* An effective approach may not be precise due to many *false prediction*s, which alert the upcoming SLA violations that have not occurred in the end[2]. A false prediction will lead to unnecessary adaptations that can bring additional cost and complexity: on one hand, runtime adaptation is costly since more resources are required to identify and to execute an adaptation plan; on the other hand, the execution of a preventive adaptation plan may become the cause of new failures.

In addition to the requirements on the accuracy, the *second challenge* lies in the efficiency. An effective decision approach has to meet the following requirements:

- *Timing.* It is desirable to start the adaptation as early as possible: late decisions are usually precise but less useful, since the best adaptation opportunities might be missed, and the benefit of preventive adaptation is diminished.
- *Efficiency.* The decision approach must be efficient (in terms of time consumption for drawing adaptation decision) in order to meet the critical time constraint at runtime.

5.1.3 Our Approach: A Two-Phase Online Prediction Approach

In response to all above-mentioned challenges, we propose a two-phase online prediction approach. As shown in Figure 5.1, an adaptation decision is proactively determined through the following steps.

- **Start.** Listen to the events emitted from the *Monitor* component, once the execution of a task t_i is completed, the two-phase online prediction algorithm is started.
- **Step 1: check the completion of the execution.** If the execution of workflow has not yet been finished, go to step 2; otherwise, the algorithm is terminated with the *silence* state, which means that no output will be generated and thus no adaptation plan will be proactively executed.
- **Step 2: evaluation of the end-to-end QoS.** The values of all QoS attributes defined in the global SLA (e.g. execution time) are re-evaluated based on the monitored data and the estimation of the execution in the future.
- **Step 3: estimation of global SLA violation.** From each QoS dimension, the evaluated value is compared with the target value defined in the global SLA. If a violation is tent to happen, a suspicion of SLA violation is reported and go to step 4; otherwise, the algorithm ends with the state *silence*.

[1] In the context of our discussion in this chapter, the *failure* refers to the degradation of the end-to-end QoS compared to the expected values.

[2] As we introduce later, a prediction approach can be evaluated by deactivating the execution of preventive adaptation actions.

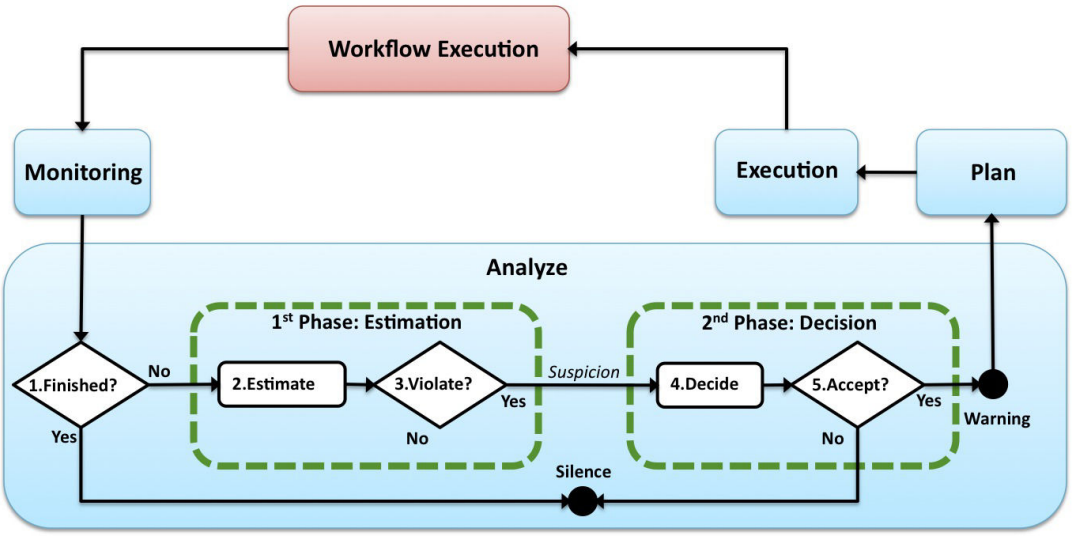


Figure 5.1: Two-Phase Online Prediction Approach

- **Step 4: evaluation of the suspicion.** A suspicion of SLA violation will not directly trigger the proactive adaptation process. Instead, the trustworthy level of the suspicion is evaluated for a more precise decision on whether to accept or to neglect this suspicion.
- **Step 5: decision for proactive adaptation.** If the suspicion is accepted, our approach terminates with the *warning* state by predicting an upcoming SLA violation and drawing the decision for identifying and executing proactive adaptation plans. Otherwise, the algorithm ends with silence state.

The core of our approach is two-phase evaluations: the *estimation phase* (step 2, 3) evaluates whether the global SLA is tent to be violated and the *decision phase* (step 4, 5) evaluates how likely the suspected violation will actually occur. An additional phase of evaluation can bring higher accuracy in making adaptation decisions since all inaccurate early suspicions are neglected in the decision phase. As a proof of concept, our discussion will focus on the execution time in the rest of this chapter. We are going to illustrate how to use the two-phase online prediction approach to accurately forecast a probable delay when the execution of SBA completes. In the following, Section 5.2 and Section 5.3 will highlight respectively the estimation phase and the decision phase. Finally, our approach will be evaluated in Section 5.4

5.2 Estimation Phase

As presented in Section 1.1.3, the end-to-end (global) execution time of an SBA instance (noted by q_t^G) depends on the local execution time of each task t_i, denoted by $q_t^L(t_i)$. However, the estimation of q_t^G for a *running* SBA instance is much more complicated since q_t^L remains unknown for some of the tasks. For a running SBA instance, the runtime state of a task can be classified into two categories:

- *Completed*, referring to the tasks whose executions have already been completed (the corresponding constituent services have already replied). For each completed task t_c, the local execution time equals to its measured execution time (noted by $q_t^M(t_c)$),

defined as the real time consumption for executing task t_c ($q_t^L(t_c) = q_t^M(t_c)$). We assume that $q_t^M(t_c)$ is known at the moment of estimation by using event-based monitoring techniques [89].

- *Uncompleted*, including the tasks that are being executed, adapted as well as the ones whose executions have not been started yet. The local execution time of an uncompleted task t_u is unknown at the moment of estimation. Accordingly, in order to estimate q_t^G of a running SBA instance, the first step is to estimate the probable execution time for each uncompleted task, noted as $q_t^E(t_u)$. Then, $q_t^E(t_u)$ is used as the local execution time to compute the global execution time of this running SBA instance ($q_t^L(t_u) = q_t^E(t_u)$).

In this section, we first present both static and dynamic methods to estimate local execution time for each uncompleted task and then we introduce an efficient tool for rapid estimation of the global execution time at runtime.

5.2.1 Estimation of Local Execution Time

The local execution time of a task t_u depends on how fast the corresponding constituent service (noted by $S(t_u)$) responds. Some research work [89] propose to use arithmetic mean value of the last n measured response time of $S(t_u)$ as $q_t^E(t_u)$. However, the performance of this method is largely affected by outliers. Suppose that the last 10 measures of response time are: 940ms, 1,020ms, 1,050ms, 1,000ms, 970ms, 1,100ms, 1,020ms, 24,060ms, 960ms, 980ms. Obviously, there is an outlier (24,060ms) and the cause can be manifold (e.g. temporary network congestion). In this case, the arithmetic mean equals to 3,310ms, which cannot properly reflect the probable response time of $S(t_i)$. In addition, this method cannot be used when there is no (sufficient) historical information. For example, in the context of on-demand SBA execution, a task may bind to a constituent service that has never been invoked before.

In the following, we provide both *dynamic* and *static* methods for estimating the local execution time of an uncompleted task t_u. We assume that SBA provider has recorded the response time of all constituent services that have been invoked before.

Dynamic estimation of local execution time. Dynamic method can be applied when $S(t_u)$ has been selected and invoked in the past executions of SBA. First of all, it looks into the past records for the information about the response time of $S(t_u)$. If enough historical information is found, the approaches presented in [78] can be used to firstly detect and remove the outliers. Then, the arithmetic mean is computed as $q_t^E(t_u)$. A more efficient alternative solution is to directly use the median value of the last n measures. In the previous example, the median value is 1 000ms: five measures are less or equal to it and five are greater, including the outlier. Compare to the arithmetic mean (3,308ms), the median value can better reflect the possible response time in the near future.

Static estimation of local execution time. In case of no (sufficient) historical information about $S(t_u)$, the static method is automatically activated. Instead of dynamically computing $q_t^E(t_u)$ based on the historic information, static method uses fixed value according to the execution state of each uncompleted task.

- If t_u is being executed and its execution has already taken longer time than the expected value defined in the local SLA (noted as $q_t(t_u)$), the real execution time

$q_t^R(t_u)$ (defined as the real time consumption since the start of the execution of t_u) is used as the estimation ($q_t^E(t_u) = q_t^R(t_u)$).

- Otherwise, the target value defined in the local SLA is directly used as the estimation ($q_t^E(t_u) = q_t(t_u)$). In this case, it is straightforward to trust the service provider.

5.2.2 Estimation of Global Execution Time

After the estimation of local execution time for uncompleted tasks, the local execution time of each task can be obtained according to Formula 5.1:

$$\text{Local execution time} : q_t^L(t) = \begin{cases} q_t^M(t); & if\ t \in \{completed\ tasks\} \\ q_t^E(t); & if\ t \in \{uncompleted\ tasks\} \end{cases} \tag{5.1}$$

The local execution time of a completed (or an uncompleted) task equals to the measured (or estimated) response time of the corresponding constituent service. By this means, the global execution time of a running SBA instance (q_t^G) can be estimated by aggregating local execution time of each task along all the execution paths, as we have introduced in Section 1.1.3. q_t^G is then compared with the expected response time of SBA defined in the global SLA (noted as $gqos_t$). If q_t^G is greater than $gqos_t$, a suspicion of global SLA violation (noted by S) is reported. Otherwise, the estimation phase is terminated with the silence state and no output will be generated. The output of the estimation phase follows the Formula 5.2.

$$\text{output} : \begin{cases} S =< T_D, TS >; & if\ q_t^G > gqos_t \\ Null; & otherwise \end{cases} \tag{5.2}$$

A suspicion S is expressed by a *suspicion tuple* noted by $< T_D, TS >$. It includes two attributes. Firstly, T_D measures the degree of the estimated delay with respect to the global execution time. As indicated by Formula 5.3, T_D refers to the ratio of the estimated delay to the expected global execution time.

$$T_D = \frac{q_t^G - gqos_t}{gqos_t} \tag{5.3}$$

Then, TS is the *time stamp* by which this suspicion tuple is generated. It is measured by the percentage (in terms of the execution time) of the execution that have completed at the moment of estimation. The computation of TS follows Formula 5.4, where AT refers to the accumulated execution time of this SBA instance and $gqos_t$ is the expected global execution time defined in the global SLA. TS can reflect the degree on the completion of a workflow execution.

$$TS = \frac{AT}{gqos_t} \tag{5.4}$$

For the reason of simplicity, we name T_D as the *estimated delay* and TS as the *completion degree* in the rest of our discussions. By this means, once a suspicion of global SLA violation is reported, the suspicion tuple is evaluated in the decision phase in order to decide whether or not to start proactive adaptation, as presented later in Section 5.3.

More efficient estimation of global execution time. However, the above-mentioned estimation approach can hardly meet the requirement on the efficiency since runtime aggregation of QoS is costly and time-consuming, especially for complex and unstructured workflows. After the completion of a task t_i, the $q_t^L(t_i)$ can be updated by replacing $q_t^E(t_i)$ by $q_t^M(t_i)$. Accordingly, the global execution time is required to be re-estimated, even though most of tasks have not updated their local execution time. In the following, we introduce the Program Evaluation and Review Technique (PERT) [57] as an efficient tool for rapid runtime estimation of global execution time. PERT was originally developed for planning, monitoring and managing the progress of complex projects. In our approach, PERT is used to monitor the workflow execution and to facilitate decision making. For each task t_i, it maintains the following additional information:

- The Expected Start Time (EST): $T_E(t_i)$. This is the expected time[3] by which the execution of task t_i can start in case that all its precedents task can accomplish on schedule.
- The Latest Finish Time (LFT): $T_L(t_i)$. This is the latest time by which the execution of task t_i has to complete without causing the delay of the ongoing execution instance in case that all its succeeding task can accomplish on schedule.
- The *slack* time: $S(t_i)$. This is the maximum acceptable delay during the execution of task t_i.
- The critical path CP. A *path* is a sequential execution of tasks from the beginning to the end of a workflow. CP is one of the execution paths with all tasks having the least slack time.

All the information is then presented on an X-Y chart. The X-axis represents the accumulated execution time, while the Y-axis is the list of tasks that are composed in the workflow. Each task t_i is represented by a single horizontal bar which starts from $T_E(t_i)$ and ends with $T_L(t_i)$. The length of the bar corresponds to the maximum acceptable duration for executing a task, under the assumption that all the other tasks complete on schedule. A bar is composed of two parts, the solid part represents $q_t^L(t_i)$ and the hollow part reflects $S(t_i)$. Finally, the tasks on the critical path are marked by a star ($\star$). A concrete example of PERT chart will be provided later on.

Construction of PERT. The PERT chart is initially constructed based on both global and local SLAs by the following steps:

1. First of all, the T_E is computed for each task t_i. The computation of $T_E(t_i)$ follows the Formula 5.5. First of all, obviously, for the first task (start) t_s, $T_E(t_s) = 0$. In the following, the computation of T_E is performed in sequential order along each execution path. If a task belongs to multiple execution paths which lead to different values, the maximum value will be used (e.g. a syntonization task can start only when all incoming branches have completed the execution).

$$T_E(t_i) = \begin{cases} 0; & t_i = t_s \\ max\{T_E(t_j) + q_t^L(t_j) | t_j \in \{direct\ precedents\ of\ t_i\}\}; & t_i \neq t_s \end{cases} \quad (5.5)$$

[3]The time used in this chapter is logical time, which is measured by the accumulated time consumptions since the start of the execution.

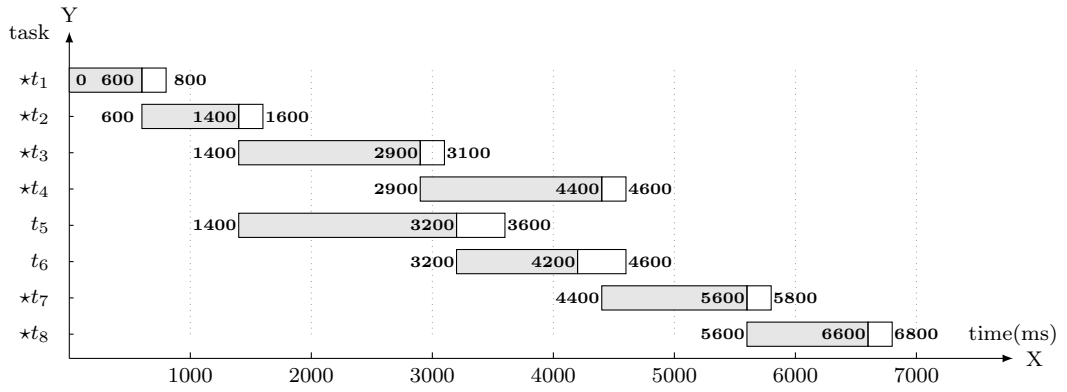


Figure 5.2: PERT Chart

2. Then, the T_L of each task is computed following Formula 5.6. First, for the last task (end) t_n, $T_L(t_n)$ equals to the expect global execution time defined in the global SLA ($gqos_t$). In the following, the computation of T_L is performed in the reverse order along each execution path. If a task belongs to multiple execution paths which lead to different values, the minimum value will be used (e.g. an and-split task has to be completed without delaying all outgoing branches.).

$$T_L(t_i) = \begin{cases} gqos_t; & t_i = t_n \\ min\{T_L(t_k) - q_t^L(t_k) | t_k \in \{direct\ successors\ of\ t_i\}\}; & t_i \neq t_n \end{cases} \quad (5.6)$$

3. Having both $T_E(t_i)$ and $T_L(t_i)$, the slack time for each task can be calculated using Formula 5.7. Please note that the expected execution time of each task $q_t^L(t_i)$ is known from the corresponding local SLA.

$$S(t_i) = T_L(t_i) - T_E(t_i) - q_t^L(t_i) \quad (5.7)$$

Using the illustrative example in Section 1.1.4 (the workflow defined in Figure 1.5 with all local and global SLAs given in Table 1.4), the corresponding PERT chart is initially computed and depicted in Figure 5.2.

Update of PERT. The PERT chart can be dynamically updated in order to reflect the most up-to-date runtime execution state of a running SBA instance. For example, when the execution of a task completes, its local execution time can be updated by replacing the formerly estimated value by the really measured one. The update of local information can lead to partial reconstruction of PERT chart by recomputing some of the above-mentioned information (e.g. the completed tasks do not need to be recomputed), which can be seen as the adjustments of some bars along the X-axis. In this context, the cost in updating PERT chart is negligible. Anyhow, the update of PERT chart can be performed while the running SBA instance is waiting for the response from constituent service(s). Thus it will not bring extra time consumption to the execution of SBA instance. The update can be triggered either periodically, or based on checkpoints or events (e.g. after each "receive" event).

Use of PERT. By using the PERT chart, the estimation of global execution time is simplified, which follows the Formula 5.8.

$$\text{output}: \begin{cases} S =< T_D, TS >, \ with\ T_D = \frac{AT - T_L(t_i)}{gqos_t}; & if\ AT > T_L(t_i) \\ Null; & otherwise \end{cases} \tag{5.8}$$

When a task t_i is finished, the real accumulated execution time AT at this moment can be easily measured. If AT is greater than $T_L(t_i)$, the global execution time is estimated to be violated by $AT - T_L(t_i)$, and a suspicion tuple S=<T_D, TS> is accordingly generated (the computation of TS still follows Formula 5.4).

In the following, we provide a possible execution of the workflow defined in Figure 1.5 in order to demonstrate how to use the PERT chart defined in Figure 5.2 to efficiently estimate a global execution time.

1. First, suppose the execution of task t_1 takes 580ms. In this case, the accumulated execution time after the completion of t_1 is 580ms. Since 580ms is less than 800ms, which defined by $T_L(t_1)$, the estimation phase terminates with the silence state according to Formula 5.8.

2. Then, suppose the execution of task t_2 costs 900ms. At this moment, the accumulated execution time comes to 1480ms (580ms+900ms). Please note that the execution of t_2 was delayed by 100ms, since the expected execution time was 800ms as defined by the local SLA in Table 1.4. However, the accumulated execution time is still below the value defined by $T_L(t_2)$, therefore no suspicion of global SLA violation will be reported.

3. In the following, task t_3 and t_4 will be executed in parallel. Suppose that the execution of task t_3 completes in 1,700ms. This time, the accumulated execution time equals to 3,180ms, which is greater than $T_L(t_3)$ (from Figure 5.2, we can see that $T_L(t_3)$ equals to 3,100ms). Accordingly, a violation of global SLA is suspected and the following suspicion is reported:

   $$S =< \frac{3180-3100}{6800}, \frac{3180}{6800} >=< 0.0012, 0.468 >.$$

   This suspicion tuple indicates that: after 46.8% of the execution has completed, the global execution time is estimated to take 0.12% more than formerly expected.

In this way, instead of running complex aggregation function, each estimation on the global execution time only requires a single comparison operation. Therefore, using PERT chart can greatly improve the efficiency in estimating the global execution time.

5.3 Decision Phase

Once a suspicion of global SLA violation is reported, the decision phase is activated to decide whether or not the proactive adaptation is needed. In this section, both static and adaptive decision strategies are introduced to evaluate the trustworthy level of a suspicion in order to accurately identify the need for proactive adaptation.

5.3.1 Decision Function

A suspicion of global SLA violation is determined based on the estimation of future execution. Therefore, the suspected violation may not definitely happen in the end. In order to evaluate the confidence level of a suspicion S, we introduce the concept of *reliability of*

suspicion, denoted as τ_s. τ_s is defined by a positive real number: greater value indicates a stronger suspicion of global SLA violation that is more probably to occur in the future. The value of τ_s is determined by the two attributes of the corresponding suspicion tuple, namely the estimated delay (T_D) and the completion degree (TS). Generally speaking, we have the following common knowledges:

1. Knowledge K_1: early suspicions are less reliable, since the global SLA violations are suspected based more on the estimated values than the measured ones. By contrast, along with the execution, as more and more local execution time have been measured, a late suspicion is considered as more reliable.

2. Knowledge K_2: greater estimated delay refers to a more reliable suspicion. Since it reflects a worse runtime execution state, the violation of global SLA is more likely to happen.

Take the following three suspicions for example:

$$S_1=<0.082,\ 0.125>,\ S_2=<0.082,\ 0.561>\ \text{and}\ S_3=<0.015,\ 0.561>$$

First, S_1 and S_2 have the same estimated delay with different completion degrees. According to the knowledge K_1, the delay estimated by S_2 is more likely to happen so that $\tau_s(S_2)$ should be greater than $\tau_s(S_1)$. Second, S_2 and S_3 are generated at the same time with different estimated delays. Followed by the knowledge K_2, S_2 is stronger than S_3 and thus it has greater reliability.

However, this knowledge is too abstract for SBA providers to draw decisions for proactive adaptations, since they can only roughly tell whether a suspicion is strong or weak (compared to another suspicion). In our approach, we aim to express these abstract knowledges by defining an *evaluation function* $\mathscr{F}$, which associates τ_s with both T_D and TS, denoted by:

$$\tau_s(S) = \mathscr{F}(T_D, TS) \tag{5.9}$$

By this means, once a suspicion of global SLA violation is reported at runtime, its reliability can be calculated. Meanwhile, an SBA provider is required to specify a *threshold of reliability*, noted as ρ_w, to express his/her minimum acceptable expectation on the reliability of suspicion. A suspicion S will be accepted if $\tau_s(S)$ is greater than ρ_w; otherwise, it will be neglected. Therefore, a suspicion of global SLA violation will lead to a real decision for proactive adaptation if and only if the Formula 5.10 is satisfied.

$$\mathscr{F}(T_D, TS) > \rho_w \tag{5.10}$$

Since ρ_w is a constant given by the SBA provider at design time, Formula 5.10 can be thus rewritten to Formula 5.11, with $\mathscr{F}_{\mathscr{D}}(T_D, TS) = \mathscr{F}(T_D, TS) - \rho_w$:

$$\mathscr{F}_{\mathscr{D}}(T_D, TS) > 0 \tag{5.11}$$

Formula 5.11 is called the *decision function*(*decision curve*). Since each suspicion is described by two attributes, it can be represented by a point on an X-Y 2-dimensional coordinate plane, with X-axis measuring TS and Y-axis representing T_D. In this context, the Formula 5.11 outlines the *decision area* on the X-Y coordination plane[4]. By this

[4]In this scenario, since both TS andT_D can be only positive real values, our discussion focus on the first quadrant.

means, when an suspicion is reported, if the corresponding point belongs to the decision area, the final decision for adaptation will be made; otherwise, it will be neglected.

In the following, we provide three static strategies and an adaptive strategy to define decision functions. *Static strategy* relies on an SBA provider's experiences and knowledges to explicitly specify a decision function that expresses the relationship between the maximum acceptable estimated delay with respect to the completion of execution. By contrast, *adaptive strategy* uses machine learning techniques to build a classifier that learns the knowledge from past suspicions in order to predict whether or not the Formula 5.11 can be satisfied once a suspicion is reported.

5.3.2 Static Decision Strategies

An SBA provider can define the decision function based on his/her knowledges or experiences. In the following, we are going to present three different types of decision functions that can be applied in different execution contexts.

Qualitative strategy. The idea of qualitative strategy follows the knowledge K_1, since early suspicions are considered as inaccurate, they all will be neglected regardless how large the estimated delay is; by contrast, all late suspicions, even with small estimated delays, will be accepted. The SBA provider has to specify a threshold of completion, noted as δ_t, which tells an early suspicion from a late one: a suspicion with TS greater than δ_t will be considered as a late one with a high level of reliability, and thus it will be accepted; otherwise, it will be neglected. Using this strategy, the decision is made based on only TS whereas T_D is not taken into consideration. Accordingly, the decision function is defined as follows:

$$\mathscr{F}_{\mathscr{D}}{}^{qual}(T_D, TS) = TS - \delta_t > 0 \tag{5.12}$$

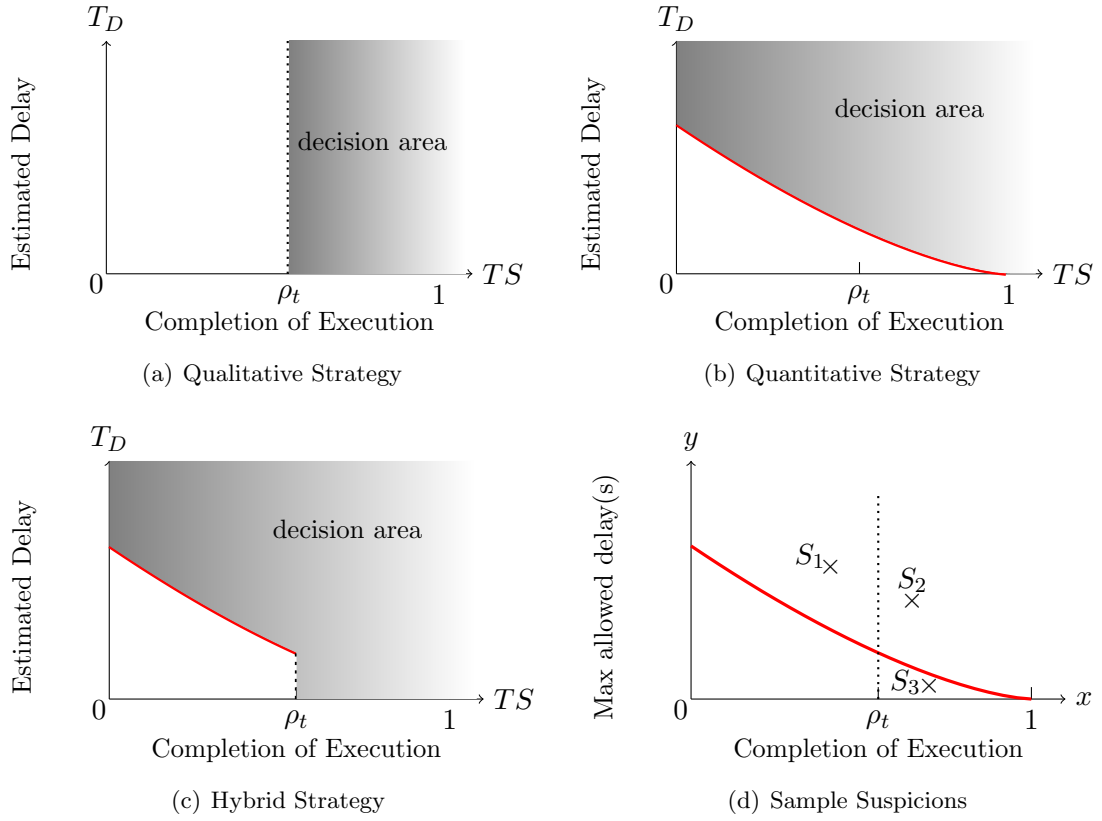
The decision area of the qualitative strategy is defined by Formula 5.12, which is depicted by the shaded area in Figure 5.3(a). Figure 5.3(d) illustrates the three sample suspicions provided in Section 5.3.1. Using qualitative strategy, S_1 is outside the decision area so that it is considered as an inaccurate early decision and it will be neglected; whereas both S_2 and S_3 will be accepted.

Quantitative strategy. The main limitation of qualitative strategy is that the adaptation cannot be triggered until a certain percentage of workflow has been executed, despite the fact that a huge delay may arise at the beginning of the execution, such as S_1. In order to react to such problem as early as possible, quantitative strategy evaluates a suspicion with the consideration of both TS and T_D. In this case, the SBA provider specifies a decision function based on his/her own experience. As a proof-of-concept example, a possible quantitative decision function is given by Formula 5.13.

$$\mathscr{F}_D^{quan}(T_D, TS) = T_D - (1 - TS)^{\frac{3}{2}} * p > 0, with\ p = 0.05 \tag{5.13}$$

The decision area is illustrated by the shaded area above the curve outlines in Figure 5.3(b). Accordingly, suspicion S_1 and S_2 will be accepted whereas S_3 will be neglected.

Hybrid strategy. The quantitative strategy may not accept a suspicion with slight delays when the execution is approaching to the end, such as S_3. In some cases, since most part of workflow has been executed, a slight delay can also finally lead to a great



(a) Qualitative Strategy

(b) Quantitative Strategy

(c) Hybrid Strategy

(d) Sample Suspicions

Figure 5.3: Comparison of Static Strategies

penalty due to global SLA violation. The hybrid strategy is more critical: by specifying δ_t, if TS is greater than δ_t, the qualitative strategy is applied. That is to say, S will be absolutely accepted. Otherwise, the quantitative strategy is applied. Accordingly, its decision function follows Formula 5.12 when $TS < \delta_t$ and follows Formula 5.13 otherwise.

The decision area of hybrid strategy is depicted in Figure 5.3(c). We can see that the hybrid strategy is more strict than both qualitative and quantitative strategies, since its decision area covers the decision area of both qualitative and quantitative strategies. Using hybrid strategy, all three sample suspicions in Figure 5.3(d) will be accepted.

5.3.3 Adaptive Decision Strategy.

Static decision strategies are useful when insufficient historical information is available. However, from the long-run perspective, it has the following limitations: first of all, it is a challenging task for SBA providers to manually identify a suitable decision function based on his/her past experiences: sometimes such experience is hard to express using a regular function. Additionally, once defined, the evaluation function cannot be self-adjusted (but can be manually modified by SBA providers) in order to automatically improve the quality of decision.

The adaptive strategy models adaptation decision as a classification problem. First of all, in the *training phase*, no adaptation plan will be actually identified and executed. Therefore, the *correctness* of a suspicion (noted by C_S) can be evaluated when the relative execution of workflow terminates: if the global SLA is actually violated in the end, this suspicion is proved as correct (C_S=*true*); otherwise, it is marked as a false one (C_S=*false*).

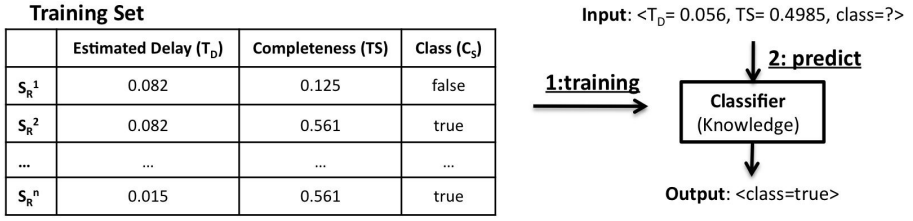


Figure 5.4: Adaptive Decision Strategy

By this means, after an execution of SBA instance completes, all the reported suspicion tuples can be transformed into a set of[5] *suspicion records*. A suspicion record is described by two numeric attributes (T_D and TS) and a categorical attribute (C_S) that is defined as the *class* (can be either true or false), denoted as follows:

$$S_R =< T_D, TS, C_S >$$

In the training phase, an SBA is executed N times, and all the collected suspicion records are organized into a training dataset, as illustrated in Figure 5.4. The dataset is often depicted as a table, with each row representing a suspicion record.

Based on machine learning technique [43], a classifier[6] is built to progressively learn the knowledge from past experiences, namely the dataset. The knowledge implements the decision function by means of either a set of complex functions, or a decision tree, or a number of rules (depending on different implementation of classifier). Compared to static strategies, the decision area of the adaptive strategy is usually irregular (sometimes, it is impossible to outline its decision area on an X-Y coordinate plane).

Later in the *prediction* phase, once a suspicion is reported, the corresponding suspicion tuple is used as the input of the classifier. The classifier is able to determine the class of this new suspicion based on its knowledge and the attributes of this suspicion . If it is classified as correct (C_S=true), it means that the Formula 5.11 is satisfied and the adaptation decision is accordingly made; otherwise, this suspicion is neglected. The classifier is required to be retrained (update the knowledge) to improve the prediction quality. The retraining can be carried out in one of the following ways: 1) after every N predictions, 2) periodically (after a fixed duration), 3) on-demand by the SBA provider.

In order to ensure the prediction accuracy, the dataset is required to be cleaned in order to detect and correct (or remove) corrupt (or inaccurate) suspicion records from the training dataset. Due to the limited space, the topic of data cleansing is out of the scope of this dissertation. Interested readers can refer to [43].

5.4 Evaluation

Our approach is evaluated and validated by a set of experiments built on a realistic simulation model, since real implementation is costly, which requires to implement the entire MAPE control loop as well as dealing with some other challenging problems of on-demand SBA execution, such as service selection, or interface mediation.

[5]Since no adaptation plan will be proactive executed, an execution of SBA instance may report multiple, one, or even no suspicion, which depends on the execution.

[6]We use the existing classifiers implemented by WEKA machine learning toolkit [72].

Table 5.1: QoS Dataset

ClientIP	WSID	Time (ms)	DataSize	HTTP Code	HTTP Message
35.9.27.26	8451	2736	582	200	OK
35.9.27.26	8460	804	14419	200	OK
35.9.27.26	8953	20176	2624	-1	java.net.SocketTimeoutException: connect timed out

5.4.1 Experiment Setup: Realistic simulation model

Virtual Service

The objective of two-phase online prediction approach is to predict failures in the future that are raised from non-functional level, namely QoS degradations. As a result, for each constituent service, we are only interested in how it responds rather than what it responds. In this context, instead of invoking real-world Web services, each task is bound to a *Virtual Service* (VS). A virtual service is a Web service that does not provide any functionality: it returns what it has received. But it simulates the non-functional aspects of a service invocation (e.g. response time, availability).

We have created 100 virtual services based on the realistic QoS datasets provided by [163]. The datasets record the non-functional performances (such as response time, throughput, etc.) of a large number of real-world service invocations. The authors have invoked 100 Web services from 150 distributed service clients located all over the world. The datasets contain 150 files, where each file includes 10,000 Web service invocations performed by one service client to all 100 Web services (in this case, each Web service has been invoked for about 100 times). A sample dataset is illustrated in Table 5.1.

By using the datasets, a virtual service is created by collecting all the invocation records to the same Web service in a file. By using these records, each virtual service defines two methods:

1. *simulate*() randomly selects one of the past records to simulate the non-functional aspects of an invocation.

2. *getExpectedRT*() determines the expected response time by specifying a percentage threshold ϕ, which indicates the percentage of past invocations which can respond within the expected value. In our experiments, in order to create a scenario with high violation rate, ϕ is set to 0.6. In other words, an invocation to a virtual service has 40% possibility to violate local SLA.

Simulate an execution of SBA.

Our experiments are based on the experimental workflow depicted in Figure 5.5, which composes 8 tasks and two parallel execution paths. An execution of this SBA is simulated through three stages:

1. In the first stage, an SBA instance is created by binding each task t_i to a randomly selected virtual service, denoted as $vs(t_i)$. Then, the local SLA for each task t_i can be generated by completing a template with the expected response time of $vs(t_i)$ (calling the $vs(t_i).getExpectedRT()$ method). Next, the global SLA is generated by

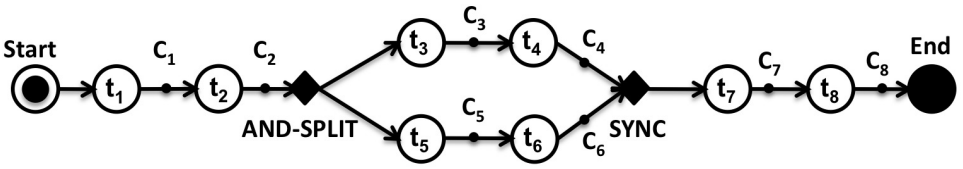


Figure 5.5: Experimental Workflow

computing the expected end-to-end execution time using aggregation functions [81]. Finally, based on both local and global SLAs, a PERT chart is constructed.

2. The second stage simulates the execution of this SBA instance. First of all, each selected virtual service $vs(t_i)$ simulates the response time (calling the $vs(t_i).simulate()$ function) as the real execution time of task t_i. Then, by running aggregation functions along all execution paths, the real accumulated execution time by which the execution of task t_i is completed can be computed, denoted as $AT(t_i)$. Next, a collection of "receive" events are created with the corresponding timestamp, denoted as $Recv = <t_i, AT(t_i)>$.

3. Finally in the third stage, these "receive" events are sorted by the timestamp and then sequentially processed: firstly, if $AT(t_i) > T_L(t_i)$, a set of predictors are activated to make adaptation decision based on different strategies. A predictor is a Java object that implements a specific decision strategy. After the prediction, the PERT chart is updated and the static method is used for estimations of local execution time.

5.4.2 Evaluation Metrics.

Contingency table metrics [127] are used to investigate how accurately a prediction approach works. Two phase online prediction approach can terminate with two possible states: warning or silence (refer to Figure 5.1). Using the contingency table, as shown in Table 5.2, a warning is defined as a *positive decision* (P), which asserts that the global SLA will be violated in the near future; by contrast, a silence is formally named as a *negative decision* (N) which decides that no adaptation was needed for the entire duration of the execution. In order to evaluate the quality of decision, no adaptation plan will really be identified and executed. For a positive decision, if a violation is really occurred in the end, it is proven to be a true positive (TP); otherwise, it is a false positive (FP). Similarly, a negative decision can be either a true negative (TN) if no violation really happens at the end of execution, or otherwise a false negative (FN). Based on the contingency table, different evaluation metrics are defined as follows:

- Accuracy (a). It is the ratio of all correct decisions to the number of all decisions:

$$a = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{T} \tag{5.14}$$

  A greater value indicates a more accurate prediction approach.

- Effectiveness (e). It is the ratio of all correct silences to the number of all silences:

$$e = \frac{TN}{TN + FN} \tag{5.15}$$

Table 5.2: Contingency Table

	Real: violated	Real: not violated	Sum
Prediction: violated (warning)	True Positive (TP) (correct warning)	False Positive (FP) (false warning)	Positive (P)
Prediction: not violated (silence)	False Negative (FN) (false silence)	True Negative (TN) (correct silence)	Negative (N)
Sum	Violations (V)	Compliance (C)	Total (T)

A greater value suggests that more violation can be successfully predicted, thus the prediction approach is more effective.

- Precision (p). It is the ratio of all correct warnings to the number of all warnings:

$$p = \frac{TP}{TP + FP} \tag{5.16}$$

A greater value implies a higher reliability for each warning, therefore the prediction approach is more precise.

- Decision Time (dt). Only positive decisions have dt. It is measured by the maximum number of tasks that have already been completed on different execution paths while the proactive adaptation decision is made.

5.4.3 Experiment 1: Evaluation of Traditional Prediction Approaches

In this experiment, the traditional prediction approaches are evaluated, namely the checkpoint based approaches and runtime verification approaches. Firstly, after each task t_i ($1 \leq i \leq 8$) in the experimental workflow, a checkpoint C_i is defined as a possible decision time point, as illustrated in Figure 5.5. 8 predictors are created based on a single checkpoint: predictor P_i ($1 \leq i \leq 8$) can decide only when the execution of workflow reaches to checkpoint C_i. Please note that the decisions of P_8 must be definitely accurate because all its decisions are determined after the execution of workflow has completed. In addition, two predictors are defined based on multiple checkpoints: P_9 is activated at both C_2 and C_7 while P_{10} can decide at either C_4 or C_6. Therefore, P_9 can be seen as the logical disjunction of P_2 and P_7 since P_9 will alert a warning of SLA violation when either P_2 or P_4 (or both of them) alerts a warning. Similarly, P_{10} is the logical disjunction of P_4 and P_6. We run 1000 executions of SBA, Table 5.3 summarizes the performance of all predictors.

Decisions based on a single checkpoint. The results show that, by using a single checkpoint, it is hard to draw both early and accurate adaptation decision. First of all, the results have proved that early decisions are less accurate. Due to the large number of FP and FN decisions, P_1 and P_2 result in lower accuracy, precision as well as effectiveness. By contrast, as the most part of workflow has been executed, P_7 performs largely better but its decisions come too late to carry out effective preventive adaptations (only one task remains unexecuted). Furthermore, the other four predictors perform poorly since they are based on the checkpoints located on two parallel execution branches. In the context of on-demand execution, each execution of SBA selects different sets of constituent services. Therefore, the critical path can be only determined at runtime due to on-demand service

Table 5.3: Experimental Results: Traditional Approaches

Metrics	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
TP	191	243	171	211	165	191	397	441	406	359
FP	156	146	59	45	61	41	39	0.0	162	79
FN	250	198	270	230	276	250	44	0.0	35	82
TN	403	413	500	514	498	518	559	520	397	480
a	0.59	0.66	0.67	0.73	0.66	0.71	0.92	1.0	0.8	0.84
e	0.43	0.55	0.39	0.48	0.37	0.43	0.90	1.0	0.92	0.81
p	0.72	0.74	0.90	0.92	0.89	0.93	0.93	1.0	0.71	0.86
dt	1	2	3	4	3	4	5	6	2.95	4.0

selection. If a task (checkpoint) is not on the critical path, it has thus a larger slack time and a longer delay can be tolerated. Therefore, from its local perspective, the execution is considered as running well and no warning will be reported. That explains why P_3, P_4, P_5 and P_6 have less FP decisions but a tremendous number of FN decisions, which lead to a poor performance on average.

Decisions based on multiple checkpoints. An additional checkpoint can bring another chance to report (either true or false) warnings of SLA violations. Take P_9 for example, if a violation is failed to be predicted by P_2, it still has chance to be alerted by P_7. Thus, compared to P_2, P_9 has greatly improved the TP number; meanwhile, only few FP decisions are additionally produced due to the high accuracy of P_7. On the other side, compared to P_7, although P_9 does get some quality degradation, but the decision time is improved by 2 execution steps. This is the same for P_{10}, compared to P_4 and P_6, both accuracy and effectiveness are significantly improved. However, as an extreme case, we assume that a predictor P_x is the logical disjunction of all predictors P_i $(1 \leq i \leq 1)$. So P_x can draw adaptation decisions at all possible checkpoints. It is similar to runtime verification techniques introduced in Section 1.4.2: whenever a deviation is estimated, the adaptation decision is made. In this case, any a predictor from P_1 to P_8 decides, P_x will also decide. Thus, P_x cannot have less FP numbers than P_1. Such a high FP number will result in a poor precision. From all above discussions, we can see that it is challenging to determine both early and accurate adaptations decisions.

5.4.4 Experiment 2: Evaluation of Our Approaches

In the second experiment, both static and adaptive decision strategies presented in this thesis are evaluated. The qualitative predictor P_{ql} implements qualitative strategy by specifying the weight threshold δ_t to 0.65. The quantitative predictor P_{qt} implements the evaluation function given in Formula 5.13. This function can tolerate greater deviation at the beginning of the execution (as shown in Figure 5.3(b)). Additionally, the hybrid predictor P_{hb} invokes P_{qt} when TS is less than 0.65, and uses P_{ql} otherwise. Finally, the adaptive predictor P_{ad} implements a set of classifiers based on WEKA machine learning toolkit [72]. All the suspicions reported in the first experiment are used to generate a set of suspicion records, which are organized in the dataset file in the following form

```
@relation knowledge
@attribute delay real
@attribute weight real
@attribute violation {true, false}
```

Table 5.4: Experimental Results: Evaluate Different Decision Strategies

Metrics	P_7	P_9	P_{10}	P_{ql}	P_{qt}	P_{hb}	P_{ad}
a	0.91	0.82	0.84	0.94	0.95	0.92	0.94
e	0.90	0.93	0.81	1.0	1.0	1.0	0.95
p	0.91	0.74	0.87	0.89	0.91	0.86	0.93
dt	5	3.04	4	4.2	3.9	3.8	3.8

```
@data
0.082, 0.125, true
...
```

During the training phase (Step 1 in Figure 5.4), all classifiers are used to learn the knowledge from the dataset, and their performances are evaluated by cross-validation. When a new suspicion is reported, the classifier with the best predictive accuracy is then used for the prediction (Step 2 in Figure 5.4). Each classifier requires at least 300 historical suspicion records (*min_training_size*=300); if the dataset contains more than 1000 records, it selects the 1000 most recent items (*max_training_size*=1000). The retraining is carried out for every 200 predictions.

Another 1000 executions are carried out to evaluate our approach. As a comparison, the three predictors with the best decision quality in the first experiment, namely P_7, P_9 and P_{10}, are also used. The performance of different predictors is shown in Table 5.4.

Comparison of different strategies. First of all, we can see that the quality of all static and adaptive decision strategies can be considered on the same level. Please note that: 1) P_{ql} decides later than the other three strategies, since it can decide only when a certain part of workflow has been executed. 2) As introduced, P_{hb} is more critical than P_{ql} and P_{qt}: it prefers FP predictions rather than FN ones. Thus it accepts more suspicions than other two two strategies, which can result in a lower precision whereas a higher effectiveness. 3) Static strategies can successfully prevent almost all of SLA violations (luckily, in this experiment, no SLA violation has been missed). Meanwhile, adaptive strategy has a lower rate of false silence (5%).

Comparison of our approach with traditional approach. Secondly, compared to checkpoint based predictors (single-step predictions), the results show that our approach can make more accurate adaptation decisions as early as possible. First of all, the decision quality of our approach can be considered on the same level as P_7. But our approach improves the decision time by more than one execution step. Secondly, P_9 decides a little earlier than our approaches (less than one execution step), but our approach has a remarkable improvement in accuracy (over 10%), effectiveness (5% higher on average) and precision (15% better). Finally, compare to P_{10}, our approaches have almost the same decision time but better accuracy (≈10%) as well as higher effectiveness (≈15%).

5.4.5 Experiment 3: Evaluations over Different Workflows

In order to evaluate the performance of our approach over different workflows, we create three fictitious workflows without real meanings:

- WF_1: a linear workflow with 9 tasks (a single execution path);

Table 5.5: Accuracy

WF	P_{ql}	P_{qt}	P_{hd}	P_{ad}
1	0.93	0.93	0.91	0.96
2	0.90	0.94	0.89	0.95
3	0.91	0.93	0.89	0.97

Table 5.6: Precision

WF	P_{ql}	P_{qt}	P_{hd}	P_{ad}
1	0.88	0.90	0.85	0.95
2	0.83	0.90	0.81	0.95
3	0.83	0.88	0.80	0.96

- WF_2: a medium workflow with 17 tasks and 6 execution paths;
- WF_3: a complex workflow with 30 tasks and 9 execution paths.

For each workflow, we firstly simulate 300 executions to initiate the dataset of suspicion records, and then 1000 simulations are carried out by using P_{ql}, P_{qt}, P_{hb} and P_{ad}. Table 5.5 and Table 5.6 summarize respectively the accuracy and the precision of different predictors. From the experimental results, we have the following observations:

1. our approach is not limited to a specific workflow and it can perform well for different kinds of workflows;
2. adaptive strategy makes decisions based on the knowledge learned from the past executions, thus its performance can always be maintained at a high level when used for different workflows;
3. the performances of static strategies depend on the predefined evaluation function: as the second experiment demonstrates, a suitable evaluation function can perform as well as adaptive strategy; otherwise, it may get a little performance degradation but it is still fairly good (compared to other approaches).

Conclusions and Perspectives

With the advent of cloud computing and Software-as-a-Service (SaaS), service-oriented computing has been adopted today by many enterprises as a flexible solution for building loosely-coupled Service-Based Applications (SBA). An SBA can be created by defining a workflow to coordinate a set of third-party Web services that are running on heterogenous platforms and distributed infrastructures. Due to this dynamic, distributed, and long-last evolving environment, flexible execution of SBA is a crucial but challenging issue from both academic and industrial perspectives. In this dissertation, we have discussed the flexible execution and adaptation of SBAs.

Contributions

This dissertation has the following contributions:

A Chemistry-Inspired Middleware for Flexible Execution of Service-Based Applications. Inspired from chemistry, service-based systems are modeled as distributed, self-organized and self-adaptive chemical systems. Services and data are described using biochemical concepts such as cells and molecules. By defining a variety of reaction rules, service selection, coordination and adaptation are described as a series of pervasive chemical reactions in the middleware. This work has the following contributions.

1. **Context-aware selection of services**[7]**.** In the middleware, an SBA can be abstractly defined at design time and suitable constituent services can be selected and integrated on the fly according to specific execution context. The selection of services can follow both local and global constraints. By introducing the concept of Partially Instantiated Workflow (PIW), service selection is performed as a recursive process to construct PIWs, from simple ones to complex ones, until the entire workflow is fully instantiated. The constructions of PIWs can be carried out in parallel rather than a sequential or one-shot process adopted by other approaches. The contribution of this work leads to the publication [63].

2. **Flexible coordination of services.** Traditional service orchestration approaches (e.g. WS-BPEL) have some limitations, such as static nature, inflexibility to express unstructured workflow, etc. I have investigated the use of chemical programming model for more flexible expression of service orchestration. Service composition is described as a compound molecule and service coordination is expressed by a

[7]This work is conducted in collaboration with CNR (Italian National Research Council)

group of reaction rules. We have shown that 1) most of BPEL constructs can be equivalently expressed using reactions rules and 2) complex workflow patterns as well as unstructured workflows can be described by defining a group of rules. This work is summarized in the publications [139], [141].

3. **Flexible adaptation for SBAs.** Due to the distributed environment, the execution of SBAs may fail or fail to meet the expected end-to-end QoS. SBA providers have to define a number of adaptation plans to react to various runtime failures. By using chemical programming model, a variety of adaptation plans can be easily expressed in terms of reactions rules with a high degree of flexibility. By using a real-life example as a proof of concept, we have demonstrated a number of chemical reactions capable of modifying the binding references or even the workflow structures on the fly in response to either functional or non-functional runtime failures. The summary of this work can be found in the publication [141], [142].

4. **Decentralized execution models for SBA.** Without centralized point of coordination, choreography model is much more complicated than orchestration model. In this dissertation, we have introduced two choreography models for self-adaptive service coordination across organizational boundaries, namely *semi-choreography* and *auto-choreography*. We have shown that the coordination rules and adaptation rules for orchestration model can be directly reused (or with little modifications) for both choreography models. The contribution of this work is presented in the publication [141].

5. **Distributed implementation of middleware.** All above-mentioned work have been validated by a number of real implementation. First, as a proof-of-concept validation to show its viability, we have implemented a real-life SBA in the middleware. All chemical services are implemented using Higher-Order Chemical Language (HOCL) and running over distributed infrastructures (Grid'5000). In the following, in order to evaluate the performance of different models, we have conducted a number of experiments by executing two experimental workflows in the middleware. Part of the experimental results have contributed to the publication [141].

A Two-Phase Approach for Accurate and Timely Adaptation Decision. SLA violations can lead to some undesirable results to SBA providers, such as reputation degradation and penalty payment. Proactive adaptation aims at executing preventive adaptation actions before SLA violations actually occur. One of the key challenges is to accurately draw adaptation decisions: on one side, it is desirable to avoid as many as SLA violations by executing preventive adaptation actions (effectiveness); on the other side, since runtime adaptation is costly, we aim to avoid unnecessary adaptations (precision). In response to this challenge, I have proposed a *two-phase online prediction* approach. Adaptation decisions can be made by predicting an upcoming end-to-end QoS degradation through two-phase evaluations. 1) First, the *estimation phase* monitors the execution of workflow based on Program Evaluation and Review Technique (PERT) and evaluates whether the end-to-end QoS degradation is probably to occur. If that is the case, a suspicion of degradation is reported. 2) Then, the *decision phase* evaluates how likely the suspected degradation will actually occur based on the knowledge learned from past experiences by using machine learning technique. This approach is validated and evaluated through a series of realistic simulations. The results have shown that the two-phase approach is able to draw not only accurate but also timely adaptation decisions compared

to other traditional prediction approaches. The presentation of this work can be found in our publication [140].

Perspectives

Our future work will concentrate on two research directions: firstly, we aim to improve the reliability and flexibility of the middleware; furthermore, we would like to extend HOCL with new features in order to improve its efficiency.

Improvement of Reliability and Flexibility of Middleware

Elastic Distributed Chemical Infrastructures. All the experiments presented in this dissertation have been conducted on distributed chemical infrastructures. However, all the nodes are pre-configured with chemical runtime to execute chemical programs. And then, the SBA providers have to manually distribute HOCL programs to different nodes. In the future, we aim to build flexible distributed chemical infrastructures by integrating a number of components capable of managing infrastructures at runtime. By this means, SBA providers are able to submit their requests for deploying and executing chemical programs but do not need to consider how these programs can be executed. New chemical virtual machines can be configured (or the idled ones can be released) according to the execution context.

Reliable Distributed Chemical Infrastructures. In this dissertation, we assume that the distributed chemical infrastructures are reliable. However, a chemical virtual machine (CVM) may physically crash or the connection between two CVMs might be interrupted. In the future, we aim to integrate some fault-tolerance mechanisms to build reliable distributed chemical infrastructures. One of our ongoing work [137] is to implement heartbeat failure detectors to notice such middleware-level failures. Each chemical service regularly sends heartbeat messages (represented by a cell of molecules) to some of its "neighbors". By this means, the downtime of a CVM can be noticed since its heartbeat is lost. Once a failure is detected, some reaction rules can be designed to replace a constituent chemical services running on this CVM by an alternative one.

Intelligent Chemical Service Ecosystem. In the future, we expect to integrate the *two-phase online prediction* approach into the chemical middleware. We aim to define a number of *learning rules* that can learn from past execution experiences and generate/update a set of *decision rules* on the fly. These decision rules represent the knowledges to make decisions for proactive adaptations. Additionally, in the current implementation, a service composition can be executed by different models. However, an SBA provider is required to specify its desirable model before the execution can starts (by activating different sets of rules). In the near future, the execution of SBA is expected to be self-adaptable to select the "best" model to execute an SBA instance according to the execution context.

Improvement of HOCL

HOCL exhibits several desirable characteristics such as autonomicity, dynamicity, evolvability, parallelism and adaptability. However, due to a higher level of abstraction, it also presents some limitations. The future work aim to improve some of these limitations, as stated below.

Efficiency. Personally, I think efficiency is the biggest obstacle for chemical programming model to be widely accepted. A chemical solution may contain a huge number of molecules, including both data and reaction rules. It is challenging to efficiently detect reactable molecules. The current HOCL implementation uses brute-force approach: for each rule, all possible combinations of molecules will be tested in order to find a possible reaction. Accordingly, it is costly in terms of execution time.

Currently, we can avoid this limitation by decomposing a big HOCL program into several smaller ones. Some reaction rules are designed to take charge of the communication among them. By this means, each chemical program can remain in a reasonable size[8] so that the reactions are able to complete faster. Accordingly, the difference in efficiency between a chemical program and other traditional approaches can be neglected.

In the future, we would like to improve the time consumption in executing HOCL programs. The main challenge lies in how to efficiently detect reactable molecules.

Security. Another important issue is the security. In the current architecture, any HOCL program can freely write a number of molecules to remote ones, and these molecules can immediately take part in the reactions in the remote chemical solutions. This mechanism promotes asynchronous communication between nodes. Nevertheless, it brings high degree of risks. This is because the reaction rule is a special kind of molecule, it can also be passed between chemical solutions. In this context, once an HOCL program receives a reaction rule from another program, such an *external* reaction rule can lead to a series of chemical reactions in its local chemical solution by consuming local molecules. It may raise some severe problems.

For example, a malicious user can define a reaction rule as follows:

$$\textbf{let } \textit{malicious\_rule} = \textbf{replace } x{::}\text{String}, ?w \textbf{ by } x$$

Once a chemical solution contains a string, the rule *malicious_rule* will remove anything else. In this case, if this malicious user write a cell including the rule *malicious_rule* and a string to a remote chemical solution, all the contents in that remote chemical solution will be removed. Considering if a chemical service receives this rule and a string, all its predefined molecules and rules will be removed. Accordingly, it cannot provide the expected functionality any more and it also becomes responseless (because the rule *send* is also removed, it losses the ability to pass the molecule(s) to other solutions).

In order to increase the security, in the future, we aim develop some analysis approaches or signature-based mechanisms to decide whether an external rule can be executed.

[8] In the current implementation of middleware, in order to ensure the performance, each program has less than 15 reaction rules and each rule requires less than 5 input molecules.

Bibliography

[1] Extensible Markup Language (XML). http://www.w3.org/XML/. 23

[2] Grid'5000. http://www.grid5000.fr. 117

[3] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. 32

[4] Simple Object Access Protocol (SOAP) 1.2. http://www.w3.org/TR/soap/. 23

[5] Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. 33, 34, 46

[6] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. http://www.w3.org/TR/wsdl20/. 23

[7] Wikipedia. http://en.wikipedia.org/wiki/Main_Page. 36

[8] Workflow Patterns home page. http://www.workflowpatterns.com. 25, 68

[9] ActiveVOS. ActiveVOS. http://www.activevos.com/. 35

[10] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26 –34, aug. 1986. 49

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

[12] A. Al-Moayed and B. Hollunder. Quality of service attributes in web services. In *2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, pages 367–372, 2010. 26

[13] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 881–890, New York, NY, USA, 2009. ACM. 32

[14] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 11–20, New York, NY, USA, 2010. ACM. 33

[15] Ayman Amin, Alan Colman, and Lars Grunske. An approach to forecasting qos attributes of web services based on arima and garch models. In *2012 IEEE 19th International Conference on Web Services (ICWS)*, pages 74–81. IEEE, 2012. 42

[16] Apache. Apache Axis2. `axis.apache.org/axis2/java/core/`. 117, 118

[17] Apache. Apache ODE. `http://ode.apache.org/`. 35

[18] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici, and Pierluigi Plebani. Paws: A framework for executing adaptive web-service processes. *IEEE Software*, 24:39–46, 2007. 41

[19] Danilo Ardagna and Barbara Pernici. Global and local qos guarantee in web service selection. In *Business Process Management Workshops*, Lecture Notes in Computer Science, pages 32–46. Springer Berlin Heidelberg, 2006. 32, 41

[20] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Softw. Eng.*, 33(6):369–384, 2007. 25, 33, 41

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010. 13

[22] Vijayalakshmi Atluri, SoonAe Chun, Ravi Mukkamala, and Pietro Mazzoleni. A decentralized execution model for inter-organizational workflows. *Distributed and Parallel Databases*, 22:55–83, 2007.

[23] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised multisets for chemical programming. *Mathematical. Structures in Comp. Sci.*, 16(4):557–580, August 2006. 54, 55, 178

[24] J-P Banâtre, Pascal Fradet, and Yann Radenac. Higher-order chemical programming style. In *Unconventional Programming Paradigms*, pages 84–95. Springer, 2005. 50, 51

[25] J.-P. Banatre, N. Le Scouarnec, T. Priol, and Y. Radenac. Towards "chemical" desktop grids. In *IEEE International Conference on e-Science and Grid Computing*, pages 135 –142, dec. 2007. 58

[26] Jean-Pierre Banâtre and Le Métayer Daniel. A new computational model and its discipline of programming. Technical report, INRIA, 1986. 51

[27] Jean-Pierre Banâtre and Le Métayer Daniel. Programming by multiset transformation. *Communications of the ACM (CACM)*, 36:98–111, 1993. 51

[28] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Métayer. Gamma and the chemical reaction model: Fifteen years after. In *Multiset Processing*, volume 2235, pages 17–44. 2001. 50, 51

[29] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Principles of chemical programming. *Electronic Notes in Theoretical Computer Science*, 124(1):133 – 147, 2005. 14, 50, 51, 178

[30] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. The chemical reaction model recent developments and prospects. In *Software-Intensive Systems and New Computing Paradigms*, pages 209–234. Springer, 2008. 50

[31] Jean-Pierre Banâtre and Thierry Priol. Chemical Programming of Future Service-oriented Architectures. *Journal of Software (JSW)*, 4(7):738–746, September 2009. 15, 58, 178

[32] Jean-Pierre Banâtre, Thierry Priol, and Yann Radenac. Service orchestration using the chemical metaphor. In *Software Technologies for Embedded and Ubiquitous Systems*, Lecture Notes in Computer Science, pages 79–89. Springer Berlin Heidelberg, 2008. 15, 58, 178

[33] JP Banâtre, P Fradet, and Y Radenac. Classical coordination mechanisms in the chemical model. *From semantics to computer science: essays in honor of Gilles Kahn. Cambridge University Press, Cambridge*, 2008.

[34] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Self-healing bpel processes with dynamo and the jboss rule engine. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–20. ACM, 2007. 41

[35] A. Barker, C.D. Walton, and D. Robertson. Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152 –166, april-june 2009. 37

[36] 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.

[37] Marin Bertier, Marko Obrovac, and Cédric Tedeschi. A protocol for the atomic capture of multiple molecules on large scale platforms. In *Distributed Computing and Networking*, pages 1–15. Springer, 2012.

[38] A.K. Bhattacharjee and R.K. Shyamasundar. Scriptorc: A specification language for web service choreography. In *IEEE Asia-Pacific Services Computing Conference (APSCC '08).*, pages 1089 –1096, dec. 2008. 37

[39] W. Binder, I. Constantinescu, and B. Faltings. Decentralized orchestration of compositeweb services. In *International Conference on Web Services (ICWS)*, pages 869 –876, sept. 2006.

[40] L. Bodenstaff, A. Wombacher, M. Reichert, and M.C. Jaeger. Monitoring dependencies for slas: The mode4sla approach. In *IEEE International Conference on Services Computing (SCC '08)*, pages 21 –29, july 2008. 42

[41] L. Bodenstaff, A. Wombacher, M. Reichert, and M.C. Jaeger. Analyzing impact factors on composite services. In *IEEE International Conference on Services Computing (SCC '09)*, pages 218 –226, sept. 2009. 42

[42] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*, volume 734. Wiley, 2011. 42

[43] Max Bramer. *Principles of Data Mining*. Springer, 2007. 137

[44] Antonio Bucchiarone, Cinzia Cappiello, Elisabetta Di Nitto, Raman Kazhamiakin, Valentina Mazza, and Marco Pistore. Design for adaptation of service-based applications: main issues and requirements. In *ICSOC/ServiceWave 2009 Workshops*, pages 467–476. Springer, 2010. 39

[45] M. Caeiro, Zsolt Németh, and T. Priol. A chemical model for dynamic workflow coordination. In *19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2011.

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

[47] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *Advanced Information Systems Engineering*, pages 13–31. Springer, 2000. 32

[48] Girish Chafle, Sunil Chandra, Pankaj Kankar, and Vijay Mann. Handling faults in decentralized orchestration of composite web services. In *International Conference Service-Oriented Computing - ICSOC 2005*, Lecture Notes in Computer Science, pages 410–423. Springer Berlin Heidelberg, 2005. 39

[49] Girish Chafle, Koustuv Dasgupta, Arun Kumar, Sumit Mittal, and Biplav Srivastava. Adaptation in web service composition and execution. In *International Conference on Web Services (ICWS'06).*, pages 549–557. IEEE, 2006. 39

[50] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 134–143, New York, NY, USA, 2004. ACM. 38

[51] David A Chappell. *Enterprise service bus*. O'Reilly media, 2009. 49

[52] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In *Web Services*, pages 168–182. Springer, 2004.

[53] Anis Charfi and Mira Mezini. Hybrid web service composition: business processes meet business rules. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 30–38, New York, NY, USA, 2004. ACM. 47, 48, 49

[54] Anis Charfi and Mira Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344, September 2007. 49

[55] Lin Chen, Minglu Li, and Jian Cao. Eca rule-based workflow modeling and implementation for service composition. *IEICE Trans. INF. & SYST., Special Section on Parallel/Distributed Computing and Networking*, 2006. 48

[56] Lin Chen, Minglu Li, Jian Cao, and Yi Wang. An eca rule-based workflow design tool for shanghai grid. In *2005 IEEE International Conference on Services Computing*, volume 1, pages 325 – 328, 2005.

[57] Desmond L. Cook. *Program Evaluation and Review Technique–Applications in Education*. 1966. 131

[58] G. Decker, O. Kopp, F. Leymann, and M. Weske. Bpel4chor: Extending bpel for modeling choreographies. In *IEEE International Conference on Web Services (ICWS)*, pages 296 –303, july 2007. 37

[59] C. Di Napoli and M. Giordano. Chemical programming for adaptation in service-based applications. In *Software Services and Systems Research - Results and Challenges (S-Cube), 2012 Workshop on European*, pages 38–39, 2012. 58

[60] Claudia Di Napoli, Maurizio Giordano, Zsolt Németh, and Nicola Tonellotto. Adaptive instantiation of service workflows using a chemical approach. In *Euro-Par 2010 Parallel Processing Workshops*, pages 247–255, 2010. 58

[61] Claudia Di Napoli, Maurizio Giordano, Zsolt Nemeth, and Nicola Tonellotto. A chemical metaphor to model service selection for composition of services. In *Proc. of the 2nd Int. Workshop on Parallel, Architectures and Bioinspired Algorithms (in conjunction with PACT'09)*, 2010.

[62] Claudia Di Napoli, Maurizio Giordano, Zsolt Nemeth, and Nicola Tonellotto. Using chemical reactions to model service composition. In *Proceedings of the second international workshop on Self-organizing architectures*, 2010. 58

[63] Claudia Di Napoli, Maurizio Giordano, Jean-Louis Pazat, and Chen Wang. A chemical based middleware for workflow instantiation and execution. In *3rd European Conference on ServiceWave (ServiceWave)*, pages 100–111, 2010. 145

[64] Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. Enabling proactive adaptation through just-in-time testing of conversational services. In *3rd European Conference ServiceWave*, pages 63–75, 2010. 42

[65] Yagil Engel and Opher Etzion. Towards proactive event-driven computing. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 125–136. ACM, 2011. 43

[66] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall Englewood Cliffs, 2004. 13

[67] Héctor Fernández. *Flexible Coordination through the Chemical Metaphor for Service Infrastructures*. PhD thesis, University of Rennes 1, 2012. 36, 58

[68] Héctor Fernández, Thierry Priol, and Cédric Tedeschi. Decentralized Approach for Execution of Composite Web Services using the Chemical Paradigm. *8th International Conference on Web Services (ICWS)*, 2010. 58

[69] Héctor Fernández, Cédric Tedeschi, and Thierry Priol. A chemistry-inspired workflow management system for scientific applications in clouds. In *IEEE 7th International Conference on E-Science (e-Science)*, dec. 2011. 58

[70] Marcus Fontoura, Toby Lehman, Dwayne Nelson, and Thomas Truong. Tspaces services suite: Automating the development and management of web services. In *In Proceedings of the 12th International World Wide Web Conference*, 2003. 49

[71] Maurizio Giordano and Claudia Di Napoli. A chemical evolutionary mechanism for instantiating service-based applications. In *Parallel Architectures and Bioinspired Algorithms*, pages 267–286. Springer, 2012. 58

[72] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 2009. 137, 141

[73] Bing Han, Jimmy Leblet, and Gwendal Simon. Hard multidimensional multiple choice knapsack problems, an empirical study. *Computers & Operations Research*, 37(1):172 – 181, 2010. 32

[74] Rainer Hauser, Michael Friess, Jochen M Kuster, and Jussi Vanhatalo. Combining analysis of unstructured workflows with transformation to structured workflows. In *Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International*, pages 129–140. IEEE, 2006. 46

[75] Qiang He, Jun Yan, Hai Jin, and Yun Yang. Adaptation of web service composition based on workflow patterns. In *International Conference on Service-Oriented Computing (ICSOC 2008)*, pages 22–37. Springer, 2008. 41

[76] Gabriel Hermosillo, Lionel Seinturier, and Laurence Duchien. Using complex event processing for dynamic business process adaptation. In *2010 IEEE International Conference on Services Computing (SCC),*, pages 466–473. IEEE, 2010.

[77] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *1st European Conference ServiceWave*, 2008. 42

[78] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85–126, 2004. 129

[79] Lican Huang, David W Walker, Yan Huang, and Omer F Rana. Dynamic web service selection for workflow optimisation. In *Proceedings of the UK e-Science All Hands Meeting*, 2005. 32

[80] IBM. IBM WebSphere Process Server. `http://www-01.ibm.com/software/integration/wps/`. 35

[81] M.C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation for web service composition using workflow patterns. In *8th International Enterprise Distributed Object Computing Conference (EDOC)*, 2004. 139

[82] Dimka Karastoyanova, Frank Leymann, Jörg Nitzsche, Branimir Wetzstein, and Daniel Wutke. Parameterized bpel processes: Concepts and implementation. In *Business Process Management*, pages 471–476. Springer, 2006. 41

[83] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[84] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, jan 2003. 39

[85] kepler. The Kepler Project - Kepler. `https://kepler-project.org/`. 35

[86] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[87] Turgay Korkmaz and Marwan Krunz. Multi-constrained optimal path selection. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 834–843. IEEE, 2001. 33

[88] Daniel Le Métayer. Higher-order multiset processing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 18:179–200, 1994. 50

[89] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *IEEE International Conference on Web Services (ICWS'10)*, july 2010. 43, 129

[90] Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime prediction of service level agreement violations for composite services. In *International Conference on Service-Oriented Computing (ICSOC/ServiceWave'09)*, pages 176–186, 2009. 43

[91] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 2010. 38

[92] Mu Li, JinPeng Huai, and HuiPeng Guo. An adaptive web services selection method based on the qos prediction mechanism. In *IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT'09)*, volume 1, pages 395–402, 2009. 42

[93] An Liu, Qing Li, Liusheng Huang, and Mingjun Xiao. A declarative approach to enhancing the reliability of bpel processes. In *IEEE International Conference on Web Services (ICWS'2007).*, pages 272–279. IEEE, 2007. 41

[94] Jiming Liu and K.C. Tsui. Toward nature-inspired computing. *Commun. ACM*, 49(10):59–64, October 2006. 14, 178

[95] Rong Liu and Akhil Kumar. An analysis and taxonomy of unstructured workflows. In *Business Process Management*, pages 268–284. Springer, 2005. 46

[96] 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.

[97] Zakaria Maamar, Djamal Benslimane, Chirine Ghedira, Qusay H. Mahmoud, and Hamdi Yahyaoui. Tuple spaces for self-coordination of web services. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1656–1660, New York, NY, USA, 2005. ACM. 50

[98] Nebil Ben Mabrouk, Sandrine Beauche, Elena Kuznetsova, Nikolaos Georgantas, and Valérie Issarny. Qos-aware service composition in dynamic service oriented environments. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 7:1–7:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc. 33

[99] 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, New York, NY, USA, 2007. ACM.

[100] Andreas Metzger, Osama Sammodi, and Klaus Pohl. Accurate proactive adaptation of service-oriented systems. In *Assurances for Self-Adaptive Systems*, pages 240–265. Springer Berlin Heidelberg, 2013. 16, 183

[101] Stefano Modafferi and Eugenio Conforti. Methods for enabling recovery actions in ws-bpel. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 219–236. Springer, 2006. 41

[102] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. Sh-bpel: a self-healing plug-in for ws-bpel engines. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53. ACM, 2006. 41

[103] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 815–824, New York, NY, USA, 2008. ACM. 41

[104] Arun Mukhija, Andrew Dingwall-Smith, and David S Rosenblum. Qos-aware service composition in dino. In *Fifth European Conference on Web Services (ECOWS'07)*, pages 3–12. IEEE, 2007. 32

[105] Christoph Nagl, Florian Rosenberg, and Schahram Dustdar. Vidre–a distributed service-oriented business rule engine based on ruleml. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '06, pages 35–44, Washington, DC, USA, 2006. IEEE Computer Society. 49

[106] Mangala Gowri Nanda. Decentralizing execution of composite web services. In *In OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–187. ACM Press, 2004.

[107] NC Narendra, Karthikeyan Ponnalagu, Jayatheerthan Krishnamurthy, and R Ramkumar. Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming. In *International Conference on Service-Oriented Computing–ICSOC 2007*, pages 546–557. Springer, 2007.

[108] Z. Nemeth, C. Perez, and T. Priol. Workflow enactment based on a chemical metaphor. In *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*, pages 127 – 136, sept. 2005. 58

[109] Z. Nemeth, C. Perez, and T. Priol. Distributed workflow coordination: molecules and reactions. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006. 58

[110] Marko Obrovac and Cédric Tedeschi. Distributed Chemical Computing : A Feasibility Study. *International Journal of Unconventional Computing*, July 2012. 59

[111] Marko Obrovac and Cédric Tedeschi. On the Feasibility of a Distributed Runtime for the Chemical Programming Model. In *14th Workshop on Advances in Parallel and Distributed Computational Models*, Shanghai, China, July 2012. 59

[112] 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)*, Shanghai, China, July 2012. 59

[113] Marko Obrovac and Cédric Tedeschi. Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models. In *14th International Conference on Distributed Computing and Networking (ICDCN 2013)*, LNCS, Mumbai, India, January 2013. 59

[114] Oracle. Oracle BPEL Process Manager. `http://www.oracle.com/technetwork/middleware/bpel/overview/index.html/`. 35

[115] Bart Orriëns, Jian Yang, and Mike.P. Papazoglou. A framework for business rule driven service composition. In *Technologies for E-Services*, Lecture Notes in Computer Science, pages 14–27. 2003. 47, 48

[116] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[117] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[118] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE'2003).*, pages 3–12. IEEE, 2003. 13

[119] Norman W Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999. 48

[120] Gabriel Pedraza and Jacky Estublier. Distributed orchestration versus choreography: The focas approach. In *Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes*, ICSP '09, pages 75–86, Berlin, Heidelberg, 2009. Springer-Verlag. 38

[121] Pegasus. Pegasus Workflow Management System. `http://pegasus.isi.edu/`. 35

[122] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46 – 52, oct. 2003. 15, 33, 179

[123] Harald Psaier and Schahram Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011. 40

[124] Yann Radenac. *Programmation chimique d'ordre supérieur*. PhD thesis, University of Rennes 1, 2007. 112

[125] F. Rosenberg and S. Dustdar. Business rules integration in bpel - a service-oriented approach. In *Seventh IEEE International Conference on E-Commerce Technology (CEC)*, pages 476 – 479, july 2005. 47, 48, 49

[126] F. Rosenberg and S. Dustdar. Towards a distributed service-oriented business rules system. In *Third IEEE European Conference on Web Services (ECOWS'2005)*, nov. 2005. 47, 49

[127] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42, 2010. 139

[128] Eric Schmieders and Andreas Metzger. Preventing performance violations of service compositions using assumption-based run-time verification. In *4th European Conference ServiceWave*, pages 194–205, 2011. 43

[129] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001. 59

[130] Sattanathan Subramanian, Philippe Thiran, Nanjangud C Narendra, Ghita Kouadri Mostefaoui, and Zakaria Maamar. On the enhancement of bpel engines for self-healing composite web services. In *International Symposium on Applications and the Internet, 2008. (SAINT'2008)*, pages 33–39. IEEE, 2008. 41

[131] Taverna. Taverna Workflow Management System. `http://www.taverna.org.uk`. 35

[132] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *IEEE International Conference on e-Science and Grid Computing*, pages 441–448, 2007. 35

[133] Wil MP Van Der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *Information and Software Technology*, 50(3):131–159, 2008. 46

[134] W.M.P. Van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003. 25

[135] Mirko Viroli and Franco Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, 180(10):1876–1892, 2010. 14, 177

[136] W3C. Web Services Choreography Description Language Version 1.0. `http://www.w3.org/TR/ws-cdl-10/`. 37

[137] Chen Wang. A qos-aware middleware for dynamic and adaptive service execution. Technical report, INRIA, 2011. 147

[138] Chen Wang. Cell clone and fusion: Increased parallelism for auto-choreography of services. Technical report, INRIA, 2013. 174, 175

[139] Chen Wang and J.-L. 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, 29 2010-july 1 2010. 58, 146

[140] Chen Wang and Jean-Louis Pazat. A two-phase online prediction approach for accurate and timely adaptation decision. In *2012 IEEE Ninth International Conference on Services Computing (SCC),*, pages 218–225. IEEE, 2012. 147

[141] Chen Wang and Jean-Louis Pazat. A chemistry-inspired middleware for self-adaptive service orchestration and choreography. In *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'2013)*, 2013. 146

[142] Chen Wang and Jean-Louis Pazat. Un middleware inspiré par la chimie pour l'exécution et l'adaptation flexible des applications basées sur services. In *RenPar'21 (Rencontres francophones du Parallélisme)*, 2013. 146

[143] Chen Wang and Thierry Priol. Hocl installation guide. Technical report, INRIA, 2009. 114

[144] Chen Wang and Thierry Priol. Hocl programming guide. Technical report, INRIA, 2009. 56, 178

[145] Hongbing Wang, Shizhi Shao, Xuan Zhou, Cheng Wan, and Athman Bouguettaya. Web service selection with incomplete or inconsistent user preferences. In *International Conference on Service-Oriented Computing*, pages 83–98, 2009.

[146] Hongbing Wang, Junjie Xu, Peicheng Li, and P. Hung. Incomplete preference-driven web service selection. In *IEEE International Conference on Services Computing (SCC '08)*, volume 1, pages 75 –82, july.

[147] Yi Wang, Minglu Li, Jian Cao, Feilong Tang, Lin Chen, and Lei Cao. An eca-rule-based workflow management approach for web services composition. In *Grid and Cooperative Computing - GCC*, pages 143–148, 2005. 48

[148] Hans Weigand, Willem jan Van Den Heuvel, and Marcel Hiel. Rule-based service composition and service-oriented business rule management. In *Regulations Modelling and Deployment (ReMoD'08)*, 2008. 48

[149] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar, and Frank Leymann. Monitoring and analyzing influential factors of business process performance. In *13th International Enterprise Distributed Object Computing Conference (EDOC '09)*, pages 141–150, 2009.

[150] Bernd W Wirtz. *Electronic business*. Gabler, 2000. 13

[151] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454 –474, 1998. 49

[152] YAWL. Yet Another Workflow Language. `http://www.yawlfoundation.org/`. 35

[153] Ustun Yildiz and Claude Godart. Towards decentralized service orchestrations. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1662–1666, New York, NY, USA, 2007. ACM.

[154] Tao Yu and Kwei-Jay Lin. Service selection algorithms for composing complex services with multiple qos constraints. In *Proceedings of the Third international conference on Service-Oriented Computing*, ICSOC'05, pages 130–143, Berlin, Heidelberg, 2005. Springer-Verlag. 33

[155] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1), May 2007. 33

[156] Weihai Yu. Consistent and decentralized orchestration of bpel processes. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1583–1584, New York, NY, USA, 2009. ACM.

[157] JohannesMaria Zaha, Alistair Barros, Marlon Dumas, and Arthur Hofstede. Let's dance: A language for service behavior modeling. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162. Springer Berlin Heidelberg, 2006. 37

[158] Liangzhao Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311 – 327, may 2004. 25, 32, 41

[159] Liangzhao Zeng, Christoph Lingenfelder, Hui Lei, and Henry Chang. Event-driven quality of service prediction. In *Service-Oriented Computing–ICSOC 2008*, pages 147–161. Springer, 2008. 43

[160] Wei Zhao, Rainer Hauser, Kamal Bhattacharya, and Barrett R Bryant. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *International Journal of Web and Grid Services*, 2(1):68–91, 2006. 46

[161] Huiyuan Zheng, Jian Yang, and Weiliang Zhao. Qos analysis and service selection for composite services. In *2010 IEEE International Conference on Services Computing (SCC)*, pages 122 –129, july 2010.

[162] Huiyuan Zheng, Weiliang Zhao, Jian Yang, and A. Bouguettaya. Qos analysis for web service composition. In *IEEE International Conference on Services Computing (SCC '09).*, pages 235 –242, sept.

[163] Zibin Zheng, Yilei Zhang, and M.R. Lyu. Distributed qos evaluation for real-world web services. In *8th International Conference on Web Services (ICWS'10)*, pages 83 –90, july 2010. 138

Part IV

Appendix

Appendix A

Example of Semi-choreography: Decentralized Coordination of SBA

In the part, we illustrate the implementation of the semi-choreography model by using the "Best Garage" as an illustrative example.

A.1 Distribution of Coordination Information

After the instantiation process (refer to Section 3.2.1), a concrete workflow is constructed, as described in Program 3.6. At this moment, the sub-solution of this running SBA instance becomes inert. Therefore, the generic coordination rule (CR) *startExeSemiChore* in the solution of CCS is able to react with this inert instance sub-solution and start the execution of service composition by the semi-choreography model. As defined in Program A.1, it firstly marks this instance tuple as "Semi-Choreography" and then puts two new CRs, namely *buildFragment* and *buildFragmentWithCCS*, into the sub-solution of this instance. Both rules will immediately become active to trigger a series of chemical reactions. The execution of the rule *startExeSemiChore* is vividly depicted in Figure A.1(a).

1. Firstly, the reaction rule *updateNeighbor* updates the neighbors of each task with the corresponding concrete binding reference, namely the signature of the selected chemical service (L.07). Each neighbor of a task is expressed by a tuple: *task_id*:*cws_signature*. Since the precedent of the first task and the successor of the last task are defined as NULL, no concrete binding information can be found by applying the rule *updateNeighbor*. Accordingly, the rule *buildFragmentWithCCS* is defined to connect both tasks with CCS. After executing both reaction rules, the concrete binding references of the neighbors for each task have been updated. Program A.2 provides a description of an updated workflow after these reactions.

2. After updating the workflow, the instance sub-solution becomes inert again and thus the CR *startDistWFFragment* in the solution of CCS is able to become active. As defined in Figure A.1, it replaces both CRs *buildFragment* and *buildFragmentWithCCS* (L.15) by the other two CRs, namely *distributeWFFragment* and *startInvocation* (L.16), which will direct a series of reactions for distributing coordination information. The execution of the rule *startDistWFFragment* is illustrated in Figure A.1(b).

Program A.1: HOCL Rules for the Distribution of Workflow Fragments

```
1 let startExeSemiChore =
2   replace "Instance":<?w>
3   by "Instance":"Semi-Choreography":<buildFragment,buildFragmentWithCCS,w>
4 let updateNeighbor =
5   replace "Task":t_1:f_1:"Concrete":<in_or_out:wfp:<neighbor,?w>,?l>:cws_1
6     "Task":t_2:f_2:"Concrete":<?l2>:cws_2
7   by  "Task":t_1:f_1:"Concrete":<in_or_out:wfp:<neighbor:cws_2,w>,l>:cws_1
8     "Task":t_2:f_2:"Concrete":<l2>:cws_2
9   if neighbor.equals(t_2)
10 let buildFragmentWithCCS =
11   replace "Task":t:f:"Concrete":<in_or_out:wfp:<Null>,?l>:cws
12   by "Task":t:f:"Concrete":<in_or_out:wfp:<"t0":"CCS-BestGarage">,l>:cws
13   if wfp.equals("Start") || wfp.equals("End")
14 let startDistWFFragment =
15   replace "Instance":"Semi-Choreography":<buildFragment=x,
       buildFragmentWithCCS=y,?w>
16   by "Instance":"Semi-Choreography":<distributeWFFragment,startInvocation,w>
17 let distributeWFFragment =
18   replace"Instance_id":inst_id,"Task":t:f:"Concrete":<?l>:cws
19   by "Instance_id":inst_id,"Task":t:f:"Distributed":<l>:cws
20   "To_send":cws:<
21     "WF_Fragment":"CCS-BestGarage":inst_id:<
22     "Task":t:f:"Concrete":<l>:cws, "Data":<>>>
23 let startInvocation =
24   replace-one "Task":t:f:"Distributed":<"In":"Start":<t_in:"CCS-BestGarage
       ">,?l>:cws,
25     "Data":<"Car":<"ID":id, ?w1>, ?w2>,"Instance_id":inst
26   by  "Task":t:f:"Invoking":<"In":"Start":<t_in:"CCS-BestGarage">,l>:cws,
27     "Data":<"Car":<"ID":id, w1>,w2>,"Instance_id":inst,
28     "To_send":cws:<
29     "Invoke_chore":"CCS-BestGarage":inst:<"operation":"diagnose","car_id":id
         >>
```

3. In the following, the reaction rule *distributeWFFragment* will generate a fragment tuple based on each task tuple (L.21-22). A fragment tuple includes the information about the corresponding task (with concrete neighbor information), the signature of the owner ("*CCS-BestGarage*"), the related instance ID (*inst_id*), as well as a data sub-solution that is used for stocking the input/output data for this instance. Then, the fragment tuple is put into a "To_send" tuple so that it could be sent to the solution of the corresponding chemical service (L.20-22). After the distribution of a workflow fragment, the runtime state of the relative task is marked as "Distributed". The distribution of workflow fragments is illustrated in Figure A.1(c).

4. Meanwhile, CCS can also distribute a number of reaction rules to all the participants for expressing the coordination information. Since a reaction rule is a special type of molecule, the distribution of coordination rule can be performed in the same way as the distribution of workflow fragments (as the limited space, the definitions of the corresponding rules will not be provided here). All the coordination rules for a specific constituent chemical service cws_i will be packaged into an "Coordination_rules" sub-solution and sent to cws_i. For example, the rule package for *CWS-SeniorT* is expressed below, which includes a number of invocation rules (e.g., *invokeT1*) as well as coordination rules (introduced later on).

Program A.2: New Instance Tuple with Updated Neighbors

```
"Instance":<
  "Instance_id":"CCS-BestGarage001",
  "Task":"t1":"diagnostic":"Concrete":<"In":"Start":<"t0":"CCS-BestGarage">,
    "Out":"XOR-split":<"t2":"CWS-RepManager","t3":"CWS-fastRepairCar">>:"CWS
        -SeniorT",
  "Task":"t2":"reparation":"Concrete":<"In":"Seq":<"t1":"CWS-SeniorT">,
    "Out":"Seq":<"t4":"CWS-payByCard">>:"CWS-RepManager",
  "Task":"t3":"reparation":"Concrete":<"In":"Seq":<"t1":"CWS-SeniorT">,
    "Out":"Seq":<"t4":"CWS-payByCard">:"CWS-fastRepairCar",
  "Task":"t4":"billing":"Concrete":<"Out":"End":<"t0":"CCS-BestGarage">,
    "In":"XOR-join":<"t2":"CWS-RepManager","t3":"CWS-fastRepairCar">>:"CWS-
        payByCard",
  "Data":<"Car":<"Type":"RARE", "ID":"ABCDEF">>
>
```

Program A.3: Coordination Rules for Aggregating Coordination Information

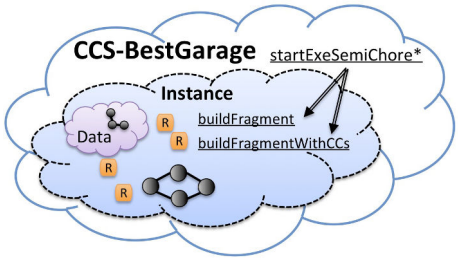
```
let updateWFFragment =
  replace "WF_Fragment":ccs1:inst_id1:<?w1>,
    "Coordination_rules":ccs2:inst_id2:<?w2>
  by "WF_Fragment":ccs1:inst_id1:<w1,w2>
  if ccs1.equals(ccs2) && inst_id1.equals(inst_id2)
```

“To_send”:“CWS-SeniorT”:<
“Coordination_rules”:“CCS-BestGarage”:“CCS-BestGarage001”:<
selectBranchCasePos_chore,selectBranchCaseNeg_chore,invokeT1,etc
>
>

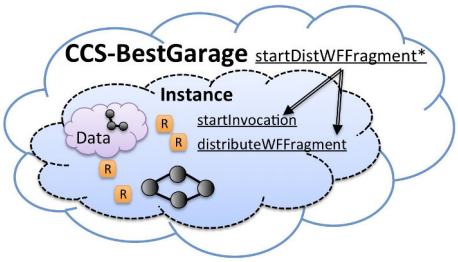
5. Since each constituent service may receive multiple coordination messages from the same SBA instance (e.g. a workflow fragment, rule packages), all these information can be aggregated in to a single one. The rule *updateWFFragment* defined in Program A.3 is able to detect a workflow fragment and a rule package with the same owner and instance ID. Then, the content of the rule package is copied into the sub-solution of the workflow fragment, as illustrated in Figure A.1(d). Up to now, each participant has a partial knowledge on the global data flow and control flow of the service composition and they can be seen as virtually connected (the network of services has been constructed).

6. The second rule *startInvocation* will generate an invocation message for the first chemical service (with the runtime state “Distributed”) in the network of services. As introduced, each semi-choreography invocation message is represented by an “Invoke_chore” tuple, which includes the signature of the proceeding chemical service, the instance ID, and a sub-solution of invocation parameter (refer to L.29 in Program A.1). In the “Best Garage” example, the invocation parameters are the same as the ones in the orchestration invocation message. Using the previous execution context, the rule *startInvocation* will generate the following message for invoking *CWS-SeniorT* chemical service.

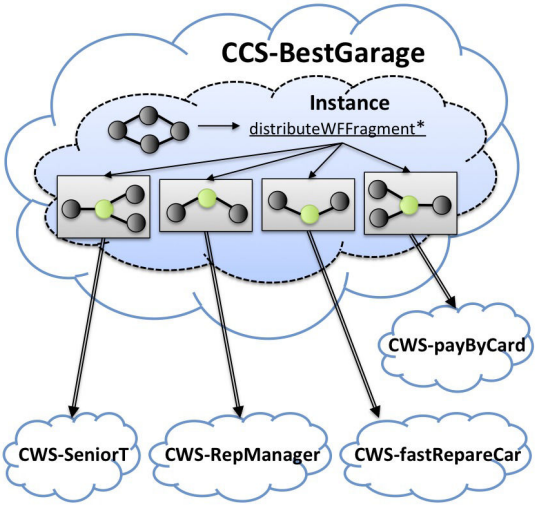
“To_send”:“CWS-SeniorT”:<
“Invoke_chore”:“CCS-BestGarage”:“CCS-BestGarage001”:<
“operation”:“diagnose”, “car_id”:id
>
>



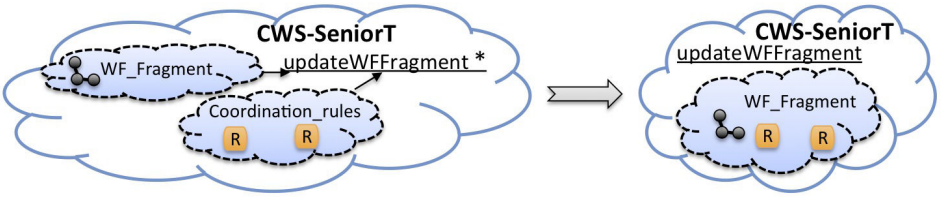
(a) Start Execution of Semi-Choreography



(b) Start Partitioning Workflow



(c) Distribution of Workflow Fragments



(d) Aggregate the Coordination Information in the Solution of CWS

Figure A.1: Illustration: Distribution of Coordination Information

A.2 Decentralized Coordination of Workflow

This initial invocation message generated by the rule *startInvocation* is sent to *CWS-SeniorT* chemical service. Similar to the dominos card, the fall of first tile of dominos will lead to a series of falling of tiles, this invocation message will lead to a series of chemical reactions in succession for executing the service composition. All the reaction rules defining this process is provided in Program A.4.

1. First of all, the arrival of a semi-choreography invocation message to a chemical service will activate the rule *getInvocationMessage*. First of all, it looks for the workflow fragment with the same instance ID. Then, all the invocation parameters

are passed into the data sub-solution of this workflow fragment (represented by p in L.04). Meanwhile, it records the signature of the preceding chemical service that has generated this invocation message (represented by *from* in L.04).

2. Once an "Invoke_chore" tuple is presented in the workflow fragment of *CWS-SeniorT*, the rule *seq_Chore_start* is activated. It requires 1) a task tuple t_i that has only one precedent with the sequence pattern 2) and a completed preceding service noted as *from* (L.29). If *from* is the exact precedent of t_i (L.31), that is to say the preceding service of this CWS has completed the execution and this chemical service can start the execution of t_i. Therefore, the runtime state of t_i is changed to "Executing" (L.30).

3. In the following, the rule *invokeT1* introduced this rule in Section 3.3 is reused to prepare an invocation request for executing task t_1.

4. The IR *invoke_diagnose_semiChore* in the solution of *CWS-SeniorT* is able to detect the creation of such an invocation request in the sub-solution of a workflow fragment. As defined in Program A.5, it reads the invocation parameters from the workflow fragment, passes them to the connector and then generates an result tuple which encapsulates the invocation response (L.05). The response can be either positive (including invocation results) or negative (runtime failures arise). In this section, we assume that reply is positive. As an example, a possible result tuple after invoking *WS-SeniorT* can be described as follows:

"Chore_Result":"CCS-BestGarage":"CCS-BestGarage001":<
"Result":<"Problem":"EASY","Time":120, "Cost":50>
>

Program A.5: Invocation Rules for Semi-Choreography (Defined by CWS)

```
1 let invoke_diagnose_semiChore =
2   replace connector,
3     "WF_Fragment":ccs:inst_id:<"Invoking":cws:<"operation":"diagnose
          ","car_id":id>,?w>
4   by  connector, "WF_Fragment":ccs:inst_id:<w>,
5     "Chore_Result":ccs:inst_id:<connector.invoke("diagnose",id)>
```

5. Such a result tuple will activate the rule *putResultChore*. It moves the invocations results to the data sub-solution of the corresponding workflow fragment (with the same signature and instance ID), and then change the runtime state of the corresponding task from "Invoking" to "Executed" (L.09).

6. Once the runtime state of task t_1 is changed to "Executed", similar to the execution of service orchestration, one of two CRs will be activated to select one of the succeeding task to continue the execution. If the problem is evaluated as easy, the rule *selectBranchCasePos_chore* decides to invoke the chemical service bound to t_2 in succession. It extends the data sub-solution with an additional destination sub-solution containing a list of succeeding services, noted as:

"Data":<?*data* >:<?*destinations* >

In the current execution context, there is only a single successor for passing the data. Meanwhile, it consumes all the other molecules in the solution of a workflow fragment (represented by a universal variable l in L.13). The other rule *selectBranch-CaseNeg_chore* is similarly defined in case of a "HARD" problem.

7. In the following, the rule *invokeSucceedingService* is able to detect a workflow fragment that contains such an extended data tuple. It will then create a semi-choreography invocation message with its own signature, instance ID and all the data for each succeeding service (L.24). This invocation message is packaged into a "To_send tuple so that it can be passed to the expected destination. Please note that, if multiple destinations are specified, multiple semi-choreography invocation message will be generated and sent. In our example, an invocation message is sent to the chemical service bound to task t_2, namely *CWS-RepManager*.

8. When all the destinations have been sent a copy of data, the destination sub-solution becomes empty. The rule *removeCompleteFragment* detects a workflow fragment with such an empty extended data sub-solution (L.26) and removes the entire workflow fragment from its solution. In our example, *CWS-SeniorT* will finally remove this workflow fragment and before it completes its job in the execution this SBA instance.

9. In the following, the execution is performed in the similar way in the solution of *CWS-RepManager*. the CR *seq_Chore_start* starts the execution by marking the runtime state of t_2 as "Executing", and finally the CR *seq_Chore_terminate* extends the data sub-solution with a single succeeding service *CWS-payByCard*. In this case, the computational results of both task t_1 and t_2 will be passed to *CWS-payByCard*.

10. The arrival of an invocation message to the solution of *CWS-payByCard* will activate the CR *simpleMerge*, which starts the execution by changing the runtime state of t_4 to "Executing". After the execution, the rule CR *seq_Chore_terminate* will sends all the data sub-solution to the succeeding service: *CCS-BestGarage*. Please note that the final result is also represented in the form of a normal semi-choreography invocation message, such as the following description.

    "To_sned":"CCS-BestGarage":<
      "Invoke_chore":"CWS-payByCard":"Best-Garage001":<
        "car_id":"ABCDEF" "Problem":"EASY",
        "pay_result":"SUCCESS","Time":120, "Cost":50>
      >

11. The arrival of such an invocation message from the chemical service that executes the last task to the CCS will activate the CR *completeExeSemiChore*. As defined in Figure A.4, it moves all the data from the invocation message to the data solution of the corresponding SBA instance and finally marked the instance as completed (L.43).

Program A.4: Coordination Rules for Semi-Choreography

```
let getInvocationMessage =
  replace "Invoke_chore":from:inst_id1:<?p>,
    "WF_Fragment":ccs:inst_id2:<"Data":<?d>,?m>
  by "WF_Fragment":ccs:inst_id2<"invoke_chore":from,"Data":<p,d>,m>
  if inst_id1.equals(inst_id2)
let putResultChore =
  replace "Chore_Result":inst_id1:<"Result":<?r>>,
    "WF_Fragment":inst_id2:<"Task":ti:fi:"Invoking":<?l>:cws,"Data":<?d>,?w>
  by "WF_Fragment":inst_id1:<"Task":ti:fi:"Executed":<l>:cws,"Data":<d,r>,w>
  if inst_id1.equals(inst_id2) && cws.equals(this.getSignature())
let selectBranchCasePos_chore =
  replace-one "Data":<"Problem":p,?result>,?w
    "Task":"t1":"diagnose":"Executed":<"Out":"XOR-Split":<"t2":cws_out,?out
       >,?in>:cws,?l
  by "Data":<"Problem":p,result>:<cws_out>
  if p.equals("EASY")
let selectBranchCaseNeg_chore =
  replace-one "Data":<"Problem":p,?result>,?w
    "Task":"t1":"diagnose":"Executed":<"Out":"XOR-Split":<"t3":cws_out,?out
       >,?in>:cws,?l
  by "Data":<"Problem":p,result>:<cws_out>
  if p.equals("HARD")
let invokeSucceedingService =
  replace "WF_Fragment":inst_id:<"Data":<?d>:<cws_out,?l>,?w>
  by  "WF_Fragment":inst_id:<"Data":<d>:<l>,w>,
    "To_send":cws_out:<"Invoke_chore":this.getSignature():inst_id:<d>>>
let removeCompleteFragment =
  replace "WF_Fragment":inst_id:<"Data":<?d>:<>,?w>,?l
  by l
let seq_Chore_start =
  replace "Invoke_chore":from,"Task":ti:fi:stat:<"In":"Seq":<t:precedent>,?
      out>:cws
  by "Task":ti:fi:"Executing":<"In":"Seq":<t:precedent>,out>:cws
  if from.equals(precedent) && stat.equals("Concrete")
let seq_Chore_terminate =
  replace "Task":ti:fi:"Executed":<"Out":"Seq":<t_out:cws_out>,?w>:cws,"Data
      ":<?d>
  by "Data":<"Problem":p,result,d>:<cws_out>
let simpleMerge =
  replace "Invoke_chore":from,"Task":ti:fi:stat:<"In":"XOR-join":<t:
      precedent,?l>,?out>:cws
  by "Task":ti:fi:"Executing":<"In":"XOR-join":<t:precedent,?l>,out>:cws
  if from.equals(precedent) && stat.equals("Concrete")
let completeExeSemiChore =
  replace "Instance":"Semi-Choreography":<"Instance_id":inst_id1, "Data":<?
      d1>
    "Task":t_i:f_i:stat_i:<"Out":"End":<?w>,?in>:cws_i,?l>,
    "Invoke_chore":from:inst_id2:<?data>
  by  "Instance":"Complete":<"Instance_id":inst_id1,"Data":<d1,data>,
    "Task":t_i:f_i:stat_i:<"Out":"End":<w>,in>:cws_i,l>,>
  if inst_id1.equals(inst_id2) && from.euqals(cws_i)
```

Appendix B

Example of Auto-choreography

In this part, we use the "Best Garage" as an example to illustrate the implementation of the auto-choreography model in the middleware.

B.1 Execution of SBA: Decentralized Coordination of Services

Once a concrete workflow is instantiated, the execution of a service composition by the auto-choreography model can be started by using the coordination rule *startExeAutoChore*. As defined in Program B.1, the sub-solution of an SBA instance is transformed to a composition cell (L.06-09), described by a "Composition_Cell" tuple. The composition cell extends an instance solution by adding the signature of the corresponding SBA provider (i.e., in our example, it equals to "*CCS-BestGarage*" L.07). Moreover, the state of the first task (noted by t_s) is changed to "Executing". Then, the entire composition cell is packaged into a "To_send" tuple (L.05-10) so that it will be passed to the solution of *CWS-SeniorT*.

1. When the composition cell arrives to the solution of *CWS-SeniorT*, the invocation rule *startInvocationAutoChore* in its solution is active to write its signature into the composition cell (L.03). As defined in Program B.2, after this reaction, the composition cell is marked as "Executing" (L.03).

2. The injection of a CWS signature will activate the reaction in the composition cell to prepare the corresponding invocation request for this CWS. In our execution context, the rule *invokeT1AutoChore* becomes active. As shown in Program B.3, its definition is similar to the invocation rule *invokeT1* introduced in Program 3.12. The difference is that 1) *invokeT1AutoChore* requires an additional input molecule: the signature of a CWS. 2) *invokeT1AutoChore* generates an additional reaction rule *forwardCompositionCell* as output molecule. 3) It requires one more condition to test: it has to verify that it is in the solution of the right constituent service that is expected to execution this task (L.11). If it is the case, similar to the rule *invokeT1*, it prepares an invocation request for executing task t_1.

3. In the following, a series of reactions are performed locally in the solution of *CWS-SeniorT* by extracting/injecting molecules from/into the composition cell. As defined in Program B.2, the coordination rule *invoke_diagnose_wf* is able to detect this invocation request from the composition cell. It will then invoke the real Web service

Program B.1: Coordination Rules for Auto-Choreography Mode

```
01 let startExeAutoChore =
02   replace "Instance":<
03       "Task":t_s:f_s:"Concrete":<"In":"Start":<Null>,?w>:cws_s, ?l
04     >
05   by  "To_send":cws_s:<
06       "Composition_Cell":<
07         "Provider":"CCS-BestGarage",
08         "Task":t_s:f_s:"Executing":<"In":"Start":<Null>,w>:cws_s, l,
09       >
10     >
11 let forwardCompositionCell =
12   replace "Task":t_n:f_n:"Executing":<?w>:cws_n, ?l
13   by  "To_send":cws_n:<
14       "Composition_Cell":<
15         "Task":t_n:f_n:"Executing":<w>:cws_n, l
16       >
17     >,
18     "Task":t_n:f_n:"Unknown":<?w>:cws_n, l,
19   if !cws_i.equals(this.signature)
20 let cleanCompositionCell =
21   replace "Composition_cell":"Ready":<?w>,?l
22   by l
23 let returnCompositionCell =
24   replace "Composition_cell":<
25     "Provider":ccs,
26     "Task":t_n:f_n:"Executed":<"Out":"End":<Null>,?p>:cws_n,,?w>,?l>
27   by  "To_send":ccs:<
28       "Result_WF":<"Task":t_n:f_n:"Executed":<"Out":"End":<Null>,p>:cws_n,w
             >,l>
29     >
30 let completeExeAutoChore =
31   replace "Result_WF":<"Instance_ID":inst_id1,?l1>,
32   by "Instance":"Completed":<"Instance_ID":inst_id1,?l1>
```

by manipulating the connector and creates a reply tuple in the composition cell after the invocation completes (L.09). Meanwhile, it modifies the state of the composition tuple to "Ready" (L.08).

4. The emergence of a reply tuple in the composition cell will activate the coordination rule *completeTask* defined in Program 3.11. As we have presented before, it puts the computational result to the "Data" sub-solution of the composition cell and then changes the state of the corresponding task from "Invoking" to "Executed". In this case, after receiving the reply, the state of task t_1 is changed to "Executed".

5. In the following, the coordination rules *selectBranchCasePos* and *selectBranchCaseNeg* are reused to select the right succeeding task to continue the execution, as we have presented in Section 3.3.1. In the following, we suppose that finally the execution is continued to task t_2 so that it is marked as "Executing".

6. The rule *forwardCompositionCell* is then activated to send a copy of composition cell to the solution of the succeeding chemical service. As defined in Program B.1, once a task is marked as executing, it creates a new composition cell by copying all the contents from the original one (L.14-16). Please note that the universal variable

Program B.2: Invocation Rules for Auto-Choreography Model (Defined by CWS)

```
let startInvocationAutoChore =
  replace "Composition_cell":<?w>
  by "Composition_cell":"Executing":<"CWS":this.getSgnature(), ?w>
let invoke_diagnose_wf =
  replace "Composition_cell":"Executing":<
      "Invoking":this.getSgnature():<"operation":"diagnose","car_id":id>,?l
    >
  by "Composition_cell":"Ready":<
      "Reply":this.getSgnature():<connector.invoke("diagnose",id)>, l
    >
  if cws.equals("CWS-SeniorT")
```

Program B.3: Invocation Rules for Auto-Choreography Model (Defined by CCS)

```
let invokeT1AutoChore =
  replace "Data":<"Car":<"ID":id, ?w1>, ?w2>,
    "Task":t1:"diagnostic":"Executing":<?w3>:cws_1,
    "CWS":cws_sol
  by  "Data":<"Car":<"ID":id, w1>, w2>,
    "Task":t1:"diagnostic":"Invoking":<w3>:cws_1,
    "Invoking":cws_1:<
      "operation":"diagnose","car_id":id>
    >,
    forwardCompositionCell
  if cws_1.equals(cws_sol)
```

l (L.12) is used to represent all the remaining molecules in the original composition cell except the task tuple t_n (explicitly given) and the rule *forwardCompositionCell* (active rule to execute this reaction). Therefore, the new composition cell is almost identical to the original one except that it does not include the rule *forwardCompositionCell*. The new copy of composition cell is encapsulate into a "To_send" tuple in order to be passed to the corresponding chemical service by the rule *send*

7. And then, the original composition cell becomes inert. So that the CR *cleanCompositionCell* in the solution of the CWS is activated. As described in Program B.1, it once a composition cell with the state of "Ready" becomes inert, it will be remove from the solution of a constituent service. Therefore, *CWS-SeniorT* will remove the original composition cell and it completes its job in this execution of SBA instance.

8. Meanwhile, the execution is continued in the solution of *CWS-RepManage* with the arrival of the composition cell. Similar to the execution of task t_1, firstly, the IR *startInvocationAutoChore* injects a signature tuple into the composition cell and then the IR *invokeT2AutoChore* is activated to prepare the invocation request for executing task t_2 and to generate a new rule *forwardCompositionCell*. In the following, the rule *invoke_repare_wf* defined in the solution of *CWS-RepManager* reads the invocation information, invokes the real-world Web service and writes a reply tuple into the composition cell. As the limited space, the definition of these reaction rules are not provided here. The reply tuple in the composition cell will activate the CR *simpleMerge* that changes the state of task t_4 to "Executing". Accordingly, the

entire composition cell is sent to *CWS-payByCard*.

9. Finally, the task t_4 is similarly executed by *CWS-payByCard* chemical service. Once the execution state of t_4 equals to "Executed", the chemical service knows that the execution of workflow has completed so that it returns the composition cell to its generator. As defined by the CR *returnCompositionCell* in Figure B.1, once the termination task of a composition cell is executed, the entire composition cell is renamed to "Result_WF" and it is packaged into a "To_send" tuple so that it can be returned to the CCS of the corresponding SBA.

10. When this composition cell arrives to the corresponding CCS's solution, the CR *completeExeAutoChore* is activated to create a completed instance tuple, which is similar to the CRs *completeExeOrch* and *completeExeAutoChore* introduced before. Please note that all the three models start from an instance tuple and end with a completed instance tuple. Thus, the execution model is transparent from the end requester.

Discussion. From the above description, we can see that almost all of coordination rules (CRs) and invocation rules (IRs) for the orchestration model can be directly reused for the auto-choreography model. However, each CWS is required to defined several additional rules to read/write the information from/to the composition cell. But fortunately, most of these rules are generic rules that can be reused by all CWSes (e.g., the rules in Program B.1). Therefore, these rules can be directly provided by the middleware when a CWS is created, which will not bring additional complexity to the providers of CWS.

Moreover, in this example, each execution of SBA can result in only one execution path. Therefore, the execution of a service composition can be easily modeled as the movement of a composition cell among all constituent services. In [138], we have discussed the reaction rules that define cell clone and cell fusion process in oder to to execute parallel branches. Due to the limited space, the details of this work is not presented here.

B.2 Decentralized Adaptation of SBA

B.2.1 Binding-level Adaptation for Auto-Choreography Model

In this part, we use *adaptation scenario 1* (refer to Section 3.3.3.1) to present the binding-level adaptation for auto-choreography model.

1. First, if the failure arises during the invocation to *WS-payByCard* real-world Web service, an error reply tuple is written into the solution of the composition cell. Such an error reply can be detected by the AR *detectFailedTask* that we have presented in Program 3.20. It will change the state of task t_4 to "Failed". Then, the AR *adaptPayByCardFailure* defined in Program 3.20 will rebind the task t_4 to CWS-payByCardPro and reset its runtime state to "Executing". Up to now, all reactions are totally the same as the orchestration model.

2. After the reaction for binding-level adaptation, the state of the failed task is reset to "Executing". So that the CR *forwardCompositionCell* defined in Program B.1 can be activated. As we have introduced in Appendix B.1, it reacts with an executing task and sends a copy of composition cell to the corresponding chemical service that has just been selected and bound to this task. In our scenario, a copy of composition cell will be sent to *CWS-payByCardPro*.

3. In the following, the original composition cell becomes inert and the reaction rule *cleanCompositionCell* in the solution of *CWS-payByCard* will be activated, as we described in Appendix B.1. Finally, the original composition cell is removed and *CWS-payByCard* finishes its job for this execution.

Discussion. From this example, we can see that the binding-level adaptation can be performed by reusing all the adaptation rules defined for the orchestration model, which brings no additional cost at design-time.

B.2.2 Workflow Level Adaptation for Auto-Choreography Model

In the following, we use the *adaptation scenario 2* introduced in Section 3.3.3.2 to present adaptation-level adaptation for auto-choreography model.

1. Firstly, similar to the orchestration model, when the execution of task t_2 cannot complete within the expected time, the runtime state of t_2 is set to "Delayed". And then, the AR *adaptaWorkflowDelay* modifies the workflow as we have presented in Section 3.3.3.2. The difference is that, for the orchestration model, the adaptation reactions are executed in the solution of *CCS-BestGarage*, whereas in auto-choreography model, the adaptation reactions are executed in the sub-solution of the composition cell, which is floated in the solution of *CWS-RepManager*.

2. After the modification of workflow, since the runtime state of task t_a is "Executing", the reaction rule *forwardCompositionCell* defined in Program B.1 becomes active. As we explained before, it generates a copy of composition cell and then sends it to the solution of *CWS-fastRentCar*. In the following, the original composition cell becomes inert (because it is waiting the result, and no reaction will take place). Please note that the rule *cleanCompositionCell* cannot be applied since it requires a composition cell with the state of "Ready" (and now, the state of the original composition cell is "Executing").

3. After a while, when task t_2 and t_a have completed, the coordination rules in both composition cells mark the state of task t_{v2} as "Executing". However, the task t_{v2} has no binding information, thus the rule *forwardCompositionCell* cannot be applied and both composition cells will stay in the solution of the corresponding chemical services. At this point, the reaction rule *invokeVirtualTask* becomes active. It changes the state of task t_{v2} to "Executed". And then some CRs mark the state of task t_4 as "Executing". As a result, both composition cells are directly sent to the solution of *CWS-payByCard*, where both cells will fuse into one to synchronize the execution. The cell fusion and the execution of parallel branches for auto-choreography is presented in [138].

Appendix C

Résumé en Français

C.1 Contexte

L'Architectures Orientées Services (SOA) sont adoptées aujourd'hui par de nombreuses entreprises car elles représentent une solution flexible pour la construction d'applications distribuées. Une Application Basée sur des Services (SBA) peut se définir comme un *workflow* qui coordonne l'ensemble des services *constitutifs* de l'application. L'exécution d'une SBA est réalisée par des appels aux services constitutifs et permet une meilleure dynamicité et adaptabilité qu'une application monolithique:

- les services constitutifs pertinents peuvent être sélectionnés et intégrés en temps réel en fonction de leur Qualité de Service (QoS);
- la coordination de services peut être effectuer soit dans une manière centralisée soit dans une manière décentralisée, en fonction du context d'exécution.
- la composition de services peut être dynamiquement modifiée pour réagir aux défaillances imprévues pendant l'exécution (par exemple, un des services constitutifs pouvant devenir indisponible).

Par conséquent, le système d'information de l'entreprise d'aujourd'hui appelle à nouveaux paradigmes pour une gestion *flexible* des applications distribuées qui enjambe les frontières organisationnelles et plates-formes informatiques hétérogènes.

C.2 Motivations

Les besoins des architectures orientées services présentent des similarités avec la nature: dynamicité, évolutivité, auto-adaptabilité, etc. Ainsi, il n'est pas surprenant que des métaphores inspirées par la nature soient considérées comme des approches appropriées pour la modélisation de tels systèmes [135].

C.2.1 Le calcul inspiré par la nature

Les gens ont régressivement appris de la nature. Par exemble, l'avion a été inventé par l'étude et de l'analyse sur les ailes de l'oiseau. Du point de vue des chercheurs des systèmes

informatiques, la nature est le plus grand système distribué, qui présente le meilleur exemple pour programmer efficacement des systèmes distribués autonomes et auto-adaptatif. Le calcul inspiré par la nature [94] vise à développer des modèles informatiques et des algorithmes inspirés par la métaphores naturelles, y compris la métaphor physique, chimique et biologique, à résoudre des problèmes pratiques et complexes dans les systèmes distribués à grande échelle.

C.2.2 Le calcul chimique

La programmation chimique est un paradigme innovant pour le calcul parallèle et autonome [29]. Inspiré par la métaphore chimique, un programme est considéré comme une *solution chimique* dans laquelle toutes les *molécules* représentent les éléments de calcul, par exemple les données. Le calcul est modélisé comme une série de *réactions* contrôlées par un ensemble de *règles*. Chaque règle précise les molécules consommées et produites par une réaction. Une règle devient active une fois que toutes les molécules nécessaires sont présentes dans la solution. La règle active peut alors modifier ou supprimer des molécules existantes ainsi que générer de nouvelles molécules. Le calcul se termine lorsque la solution passe dans un état *inerte* où aucune réaction ne peut être déclenchée. Le langage de programmation chimique d'ordre supérieur (HOCL) est un langage de programmation qui met en œuvre le modèle de programmation chimique [23, 144]. L'exécution d'un programme chimique présente les caractéristiques suivantes:

- **non-déterminisme :** la règle réagit avec les molécules choisies par hasard;
- **parallélisme :** plusieurs réactions peuvent avoir lieu indépendamment et simultanément;
- **évolution :** les réactions ne sont jamais explicitement appelées car la résultante d'une réaction peut activer des autres règles et déclencher une nouvelle série de réactions;
- **autonomie :** une règle peut manipuler d'autres règles (le programme peut donc s'auto-modifier afin de s'adapter automatiquement aux contexte d'exécution).

C.2.3 Objectives

Cette thèse vise à répondre aux défis de la construction de systèmes à base de services flexibles. Certains travaux de recherche préliminaires ont été menées pour étudier la faisabilité et la viabilité dans l'utilisation de calcul chimique pour programmer des systèmes basés sur services autonomes [32, 31]. L'*objectif* principal de cette thèse est de concevoir, de développer et d'évaluer un middleware de service en utilisant le modèle de programmation chimique pour réaliser l'exécution flexible des SBA. Afin d'atteindre cette objectif, nous allons discuter les problèmes de recherche suivants:

Comment à construire SBA dans une manière flexible? Utilisation de l'Internet comme la base de communication, de plus en plus de services sont disponibles qui peuvent fournir la même fonctionnalité sous qualités de service (QoS) différentes (par exemple le coût, le temps de réponse, etc.). En réponse à diverses exigences des utilisateurs différents, une SBA doit être construits sur demande en sélectionnant et en intégrant des services adaptés à la volée.

Comment à coordonner des services constitutifs dans une manière flexible? Orchestration et chorégraphie représentent respectivement le modèle centralisé et décentralisé pour la coordination des services [122]. Comparé avec le modèle d'orchestration, chorégraphie peut améliorer des performances (i.e. temps d'exécution). Toutefois, il apporte également des complexités et des défis supplémentaires, tels que des problèmes de tolérance aux pannes, la sécurité etc. Par conséquent, le système basé sur services de future est nécessaire de capable de choisir avec souplesse le meilleur modèle pour exécuter une composition de service en fonction de son contexte d'exécution.

Comment à réagir aux défaillances pendant l'exécution dans une manière flexible? Une instance de SBA en cours d'exécution peut échouer en raison de l'environnement d'exécution distribué, hétérogène et faiblement couplés. Par exemple, les défaillances d'infrastructure peuvent causer un service totalement responseless, qui entraînera en succession à SBA non livrable. Dans ce contexte, l'exécution de SBA est nécessaire pour être auto-adaptatif et auto-optimisation à la volée afin de réagir à ces changements dans l'environnement d'exécution.

En outre, nous ne limitons pas notre recherche seulement dans les enquêtes sur des solutions à base de l'approache chimique. Dans ce contexte, le *deuxième objectif* de cette thèse est de trouver des solutions génériques, tels que des modèles et des algorithmes, pour répondre à certains des problèmes les plus difficiles sur l'exécution flexible des SBA. Par exemple, comment à prédire et éviter l'apparition de défaillances de manière proactive au lieu de réagir passivement à eux? Ces contributions sont plus générique, qui peut être intégrée dans le middleware chimique ainsi que mis en œuvre par d'autres approches traditionnelles.

C.3 Contributions

C.3.1 Un Middleware inspiré par la chimie

Le *middleware* peut être considérée comme un "pool de services" (voir la figure 3.1) dans lequel flotte l'ensemble des abstractions chimiques des services :

- chaque service Web possède une représentation chimique au niveau du *middleware*, que nous appelons un Service Web Chimique (CWS). Il implémente une solution chimique pour encapsuler les méta-données du service Web correspondant, par exemple, la définition de son interface.
- l'application est représentée par un système chimique autonome que nous appelons Service Composé Chimique (CCS). Il implémente une solution de molécules permettant d'exprimer le *workflow* et la coordination des services;
- un ensemble de règles sont définies pour automatiser les interactions (invocations/réponses) entre les services chimiques (CWS et CCS). La transmission de messages est modélisée par le mouvement de molécule d'une solution à une autre.

C.3.1.1 Sélection de services

Dans le middleware, un provider de SBA ne décrit que le workflow d'une manière abstraite au moment de la conception. Il/elle specifie la coordination d'un ensemble de tâches abstraites avec des exigences fonctionnelles spécifiques. Ce workflow abstrait doit

être instancié par sélectionner des services constitutifs adaptés avant l'exécution peut commencer. La sélection de services est réalisée par des réactions chimiques dans le middleware en foncton des contraintes locales ou des contraintes globales. Nous présentons le concept de workflow instanciation partielle (PIW). Un PIW est une *sous-graphe structurée* d'un workflow qui doit répondre aux exigences suivantes: 1) chaque PIW a qu'une seule tâche de source (point d'entrée) et une tâche de bout (point de sortie); 2) Si la tâche de source est distincte de la tâche de bout, tous les successeurs (précédents) de la tâche de source (bout) appartiennent à ce PIW. 3) Pour toutes les autres tâches à l'exception des tâches de sources et de bout, leur précédent (s) et successeur (s) appartiennent à ce PIW. 4) Chaque tâche dans un PIW est une tâche concrète avec la référence de liaison.

Basé sur cette définition, un workflow peut être classés en quatre catégories: tâche instanciée (IT-PIW), chaîne instancié (IC-PIW), Bloque structuré instancié (ISB-PIW), workflow entièrement instancié (FIW). Par ce moyen, l'instanciation de workflow est modélisée comme un processus récursif des constructions de PIWs: premièrement, un ensemble de IT-PIWs sont construits; et puis IC-PIWs et ISB-PIWs, jusqu'à ce que l'ensemble du workflow est entièrement instancié. L'instanciation complète quand un FIW est identifié.

Cette approche présente certains avantages en comparant à des approaches traditionnelles. Tout d'abord, à la différence des approches traditionnelles, la séléction de services n'a pas besoin de transformer une description de workflow à certains modèles mathématiques ou vice versa. Donc la complexité du moment de la conception et l'exécution peut être réduite. En outre, l'instanciation et l'exécution peuvent être effectuées en parallèle car les deux processus sont décrits comme des réactions chimiques et des réactions différentes peuvent avoir au lieu simultanément. Enfin, notre approche est adaptable à l'environnement d'exécution évolution. Par exemple, au cours du processus d'instanciation, si une offre avec une meilleure qualité de service est trouvé, il peut être ajouté de manière dynamique dans la solution et ensuite participer immédiatement dans les réactions d'instanciation.

C.3.1.2 L'orchestration: le modèle centralisé de coordination de services

Après l'instanciation de workflow, cette instance de SBA peut être exécuté en coordonnant tous les services constitutifs. Le modèle d'orchestration présente la gestion de SBA d'une manière centralisé. Premièrement, CCS spécifie le flux de données par une melécule complexe. Chaque tâche est définie par une tuple de tâche, qui contient tout information de cette tâche, par exemple, son voisins (précédents et successeurs). Ensuite, un workflow est exprimée comme un *cellule* qui comprend un certain nombre de molécules de tâche, définie par un tuple de workflow. Enfin, un certain nombre de règles de réaction sont définies dans la solution de SCC (plus précisément, dans la sous-solution de chaque instance de SBA) afin de gérer l'exécution des compositions de service, par exemple, pour contrôler l'ordre d'exécution de services chimiques constitutifs. Selon les fonctionnalités différentes, toutes les règles de réaction dans la solution du CCS peuvent être classés en trois catégories.

- **Règles de coordination (CRs).** Les règles de coordination prennent en charge de coordonner l'ordre de l'exécution des tâches de workflow. Dans cette thèse, nous avons montré que tous les modèles complexe de workflow peut être exprimé en utilisant des règles de réaction.
- **Règles d'invocation (IRs).** Les règles d'invocation gérer des invocations à services constitutifs dans le middleware. Chaque tâche t_i est associée à une règle d'invocation, a noté que *invokeTi*, qui prend en charge la préparation et l'envoi d'un message d'invocation au service chimique correspondant.

- **Règles d'adaptation (ARs).** Les règles d'adaptation réagissent à des défaillances au cours de l'exécution du workflow. Les défaillances peuvent se produire soit au niveau fonctionnel (par exemple, un service constitutif tombe au panne) soit au niveau non-fonctionnelle (par exemple, une réponse tardive). Le provider de SBA peut exprimer des plans d'adaptation différents à l'aide de la définition de ARs. Nous avons montré l'adaptation au niveau de liaison (changement d'un service constitutif) ou au niveau de workflow (modification de structure de workflow) dans cette thèse.

l'orchestration de services est effectuée par une série de réactions chimiques distribués dans le middleware. D'un côté, la solution d'un CCS présente le contrôle centralisée de coordination. La plupart des règles de coordination sont génériques. Ainsi, ils peuvent être fournis directement par le middleware. En revanche, toutes les règles d'invocation sont spécifiques. Par conséquent, ils doivent être définis directement par le fournisseur SBA puisque chaque règle d'invocation doit traiter des différents paramètres d'appel ainsi il peut difficilement être générique. De l'autre côté, chaque service chimique constituant effectue respectivement une tâche simple. Une fois un tuple d'invocation arrive à sa solution, il appelle le service Web connecté. Et puis, quand le résultat est reçu, il l'encapsule et le renvoie à la solution de CCS. Tous les services constituant n'ont aucune connaissance sur la logique interne du SBA et donc ils ne participent pas à la coordination des services. Le modèle de l'orchestration est illustré dans figure 3.9.

C.3.1.3 La choréograpie: le modèle decentralisé de coordination de services

le modèle d'orchestration présente certaines limitations de performance (e.g., le temps d'exécution) à cause du point centralisé de coordination. Dans cette thèse, nous présentons également deux modèles décentralisés pour l'exécution de compositions de services dans le middleware à savoir: *semi-chorégraphie* et *auto-chorégraphie*. Le premier modème repose sur la répartition des fragments de workflow et les règles de coordination afin de réaliser la coordination des services décentralisés; tandis que la seconde modèle considère la composition de service en tant que cellule autonome, qui peut être cloné, condensé, et passe entre les solutions de tous les services chimiques constitutifs.

Le modèle semi-choréographie. Le modèle de semi-chorégraphie sépare le contrôle de la coordination du contrôle de l'adaptation. Chaque service constitutif est capable de coordonner des messages d'interaction avec les autres participants, mais il ne peut pas réagir aux défaillances d'exécution. Tous les défaillances doivent être traitées par un moteur d'adaptation centralisé, à savoir la solution de CCS. Le modèle semi-chorégraphie décrit cette collaboration de services décentralisés comme un *réseau de services*, où tous les services constitutifs peuvent être considérées comme virtuellement connecté selon le plan d'exécution global (flux de données et de flux de contrôle). Un réseau de services décrit d'une composition de service en tant que système de réseau de transport par pipeline: l'exécution d'une composition de service est modélisé comme le transport des marchandises (par exemple des gaz ou liquides) à travers un (plusieurs) tube(s), à partir de la source (tâche) à la destination (tâche).

Puisque tous les services constitutifs sont développés et gérés par des organisations différentes, la clé pour mettre en œuvre la coordination décentralisée réside dans la création du réseau de services. Dans ce contexte, l'exécution d'une composition de service est effectuée par deux étapes. Tout d'abord, l'étape *configuration* est de mettre en place le réseau de services par la distribution d'informations de coordination. Ensuite, à l'étape *exécution*, une composition de service est exécutée par une série d'interactions directes

entre les services constitutifs qui suivent le réseau de services préconfiguré. Les deux étapes de semi-choréographie sont illustré respectivement dans le figure 3.18 et le figure 3.19.

Le modèle auto-choréographie. Le modèle de semi-chorégraphie présente un degré élevé de complexité. D'un côté, son implémentation est basé sur des définition d'un plus grand nombre de règles de réaction qui décrivent une série de réactions plus complexes que le modèle d'orchestration. De l'autre côté, il est coûteux en établissement d'un réseau de services ainsi que dans l'adaptation aux défaillances d'exécution, car un grand nombre de messages de coordination doivent être générés et distribués sur le réseau.

Du point de vue de l'auto-chorégraphie, une composition de service est modélisée comme une *cellule de composition* autonome, qui encapsule un ensemble de molécules d'exprimer une composition de services autogérés et l'auto-adaptable. Dans ce contexte, une composition de service est exécutée par le mouvement de la cellule de composition parmi tous les services chimiques constitutifs. En recevant une cellule de composition, un service constitutif peut lire des données et d'information de coordination dans la cellule, puis faire le travail relatif, écrire de nouvelle information (résultats) dedans et finalement le transmettre au service(s) suivant(s). Lorsque l'exécution est terminée, la cellule entière, y compris les résultats définitifs, sera retourné au CCS.

En conséquence, l'auto-chorégraphie peut être considéré comme une *orchestration mobile*. Tous les services constitutifs peuvent être le coordonnateur à son tour. Quand une cellule de composition arrive à un CWS, toutes les règles de réaction (CRs, IRs et ARs) sont disponibles en lisant le contenu de la cellule de composition. Ainsi, chaque service constitutif a la connaissance de la totalité du flux de donnée (exprimée par des molécules), flux de contrôle (exprimé par CRs) ainsi que d'une collection de plans d'adaptation prédéfinis (exprimée par ARs). Par conséquent, la coordination et l'adaptation des services (si nécessaire) peuvent être ainsi réalisées localement dans sa solution. Le modèle d'auto-choréographie est illustré dans le figure 3.21.

C.3.1.4 Implémentation et evaluation

Infrastructures Distribuées Chimiques. Le middleware est implémenté sur une infrastructure distribuée. Tout d'abord, chaque service chimique est défini par un programme HOCL. Puis, tous les programmes HOCL que nous appelons ici une Machine Chimique Virtuel (CVM). La CVM intègre un module de communication réalisé en Java RMI qui met en œuvre la communication entre les programmes chimiques. Ce module de communication fournit une primitive *put* permettant l'écriture à distance dans une solution. Dans le prototype réalisé, le communicateur RMI est capable de localiser une solution à distance en utilisant le nom de solution et l'adresse de la CVM où est hébergée la solution.

Evaluation des molèles de coordination différents. Nous avons évalué des modèles de coordination différents par l'exécutions de deux workflow expérimentaux. Après analyser des résultats, nous avons les observations suivantes.

- Le modèle orchestration présente une faible complexité au moment de la conception et d'une moyen complexité au monment d'exécution. Il présente une faible efficacité dans l'exécution de workflow avec une échanges de données de grande taille, mais un faible coût ainsi que pour l'exécution d'adaptation exécution. Par conséquent, il apte à exécuter le workflow inter-organisationnel qui nécessite moins d'échanges de données et les défaillance peuvent survenir souvent car tous les services constitutifs peuvent être distribués largement dans le monde entier.

- Le modèle semi-chorégraphie a une complexité élevée au moment de la conception, mais il peut toujours donner un meilleur temps de réponse grâce à l'interaction directe entre les services constitutifs. Cependant, l'adaptation présente une grande complexité, surtout dans le contexte où les défaillances se produisent fréquemment. En conséquence, semi-chorégraphie est approprié pour la construction d'une compositions de services intra-organisationnelles, où quelques défaillances arrivent pas souvent. En outre, dans ce contexte, tous les fragments de workflow peuvent être pré-installé dans tous les services constitutifs car le gestionnaire de SBA a le contrôle sur tous les services constitutifs.

- Le modèle auto-chorégraphie apporte une complexité de moyenne. Toutefois, vu que la coordination et l'adaptation peuvent être effectuées directement par chaque service fonctionne, il présente une faible complexité d'exécution et une grande efficacité. Par conséquent, ce modèle est adapté pour exécuter des workflow intensif de données, où une échange de données de grande taille est requise (par exemple, les workflows scientifiques).

C.3.2 Une approche de prédiction en ligne en deux phases

Le middleware chimique a réalisé certaines des étapes importantes dans le cycle de vie de la gestion de la SBA, à partir de sa construction, à son exécution et d'adaptation. Toutefois, il ne peut que réagir aux défaillances au lieu de prévenir l'avenir des défaillances. Dans ce travail, nous étudons le problème d'adaptation *proactive* [100]. L'objectif est de garantir la qualité de service de SBA de bout en bout par l'exécution des mesures d'adaptation préventives avant les dégradations de qualité de service se produisent réellement. Car l'exécution d'adaptation est coûteux, l'un des défis principaux pour mettre en œuvre efficacement l'adaptation proactive est de assurer la précision de décision d'adaptation afin d'éviter les adaptations inutiles.

J'ai proposé une *approche de prédiction en ligne en deux phases*. Une décisions d'adaptation peuvent être prises par la prédiction d'une dégradation de QoS de-bout-en-bout dans le futur par des évaluations en deux phases. Tout d'abord, la *phase d'estimation* surveille l'exécution du workflow basé sur Program Evaluation and Review Technique (PERT) et évalue si une dégradation de QoS de-bout-en-bout est probablement de se produire. Si il est le cas, une suspicion de dégradation est rapporté. Ensuite, le *phase de décision* évalue la probabilité que la dégradation de QoS présumée se produira réellement basée sur les connaissances apprises des expériences du passé.

Basé sur une série de simulations réalistes, notre approche présente les propriétés souhaitables suivantes. 1) Il est capable de prendre des décisions d'adaptation précises pour des workflows différents: presque toutes les violations peut être prédit avec succès; en attendant, seulement quelques adaptations inutiles seront déclenchées. 2) Avec le même niveau de précision, notre approche peut décider le plus tôt que les approaches traditionelles. 3) En utilisant des stratégies statiques, notre approche peut encore prendre une décision d'adaptation exacte et opportune lorsque aucune information historique (or insuffisante) est disponible.

C.4 Publications

Les contributions présentées dans ce manuscrit ont été publiées dans plusieurs conférences internationale et nationale avec comité de lecture.

Conférences Internationale

1. Chen Wang and Jean-Louis Pazat: A Chemistry-Inspired Middleware for Self-Adaptive Service Orchestration and Choreography. In *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'2013)*. In Delft, The Netherlands (May 13–16, 2013).

2. Chen Wang and Jean-Louis Pazat: A Two-Phase Online Prediction Approach for Accurate and Timely Adaptation Decision. In *the 9th IEEE International Conference on Service Computing (SCC'2012)*. Honolulu, Hawaii, USA (June 2012).

3. Claudia Di Napoli, Maurizio Giordano, Jean-Louis Pazat and Chen Wang: A Chemical Based Middleware for Workflow Instantiation and Execution. In *the 3rd European Conference on ServiceWave (ServiceWave'2010)*: 100-111. Gent, Belgium (December 2010).

4. Chen Wang and Jean-Louis Pazat: Using Chemical Metaphor to Express Workflow and Service Orchestration. In *the 10th IEEE International Conference on Computer and Information Technology (CIT'2010)*: 1504-1511. Bradford, UK (June 2010).

Conférences Nationales

1. Chen Wang and Jean-Louis Pazat: Un Middleware Inspiré par la Chimie pour l'Exécution et l'Adaptation Flexible des Applications Basées sur Services. In *RenPar'21 (Rencontres francophones du Parallélisme)*. Grenoble, France (January 2013).

Rapports de recherche

1. Chen Wang: A QoS-Aware Middleware for Dynamic and Adaptive Service Execution. (May 2011) Available online at HAL-INRIA: *http://hal.inria.fr/hal-00794027/*.

2. Chen Wang: A Middleware Based on Chemical Computing for Service Execution - Current Problems and Solutions. (Jan. 2011) Available online at HAL-INRIA: *http://hal.inria.fr/hal-00794023/*.

Rapports techniques

1. Chen Wang, Thierry Priol: HOCL Programming Guide. Technique report (Sept. 2009). Available online in HAL-INRIA: *http://hal.inria.fr/hal-00705283/*.

2. Chen Wang, Thierry Priol: HOCL Installation Guide. Technique report (Aug. 2009). Available online at HAL-INRIA: *http://hal.inria.fr/hal-00794028/*.

C.5 Organisation du manuscrit

Le reste de ce manuscrit est organisé en trois parties.

Part I: Contexte

Dans cette partie, nous présentons l'état de l'art du contexte de nos travaux.

- **Chapter 1** introduit premièrement le paradigme du calcul orienté aux services. Ensuite, nous décrivons des problèmes de recherche et des travaux de recherche connexes.

- **Chapter 2** présente tout d'abord des approches non conventionnelles qui sont appropriées pour développer des systèmes basés sur services, tels que des systèmes basé sur règles et des systèmes basés sur *tuple space*. Ensuite, nous introduisons une approche similaire mais plus préférable connu comme la modèle de programmation chimique ainsi que d'un langage de programmation chimique d'ordre supérieur (HOCL).

Part II: Middleware inspiré par la chimie

Dans cette partie, nous présentons la première contribution de nos travaux. Nous allons présenter la conception, la réalisation et l'évaluation d'un middleware basé sur le modèle de programmation chimique pour l'exécution flexible des applications basées sur les services.

- **Chapter 3** se concentre sur la conception de middleware. Tout d'abord, la vue d'ensemble du middleware est illustré. Ensuite, nous présentons un certain nombre de règles de réactions qui décrivent la sélection de services en termes d'une série de réactions chimiques dans le middleware. Dans ce qui suit, un modèle centralisé (*orchestration*) et deux modèles décentralisés (*semi-choreography* and *auto-choreography*) sont introduites pour la coordination et l'adaptation autonome des services.

- **Chapter 4** se concentre sur l'implémentation et l'évaluation du middleware. Tout d'abord, nous analysons les performances des modèles différents en utilisant deux workflow expérimentales. Ensuite, nous présentons l'implémentation d'HOCL et les infrastructures chimiques distribués sur lequel le middleware peut être exécuté. Enfin, comme une validation pour montrer la viabilité de notre approche, nous montrons un certain nombre d'expériences par l'exécution de deux workflow expérimentales dans le middleware. Les résultats de l'évaluation prouvent notre analyser au début de cette section.

Part III: Adaptation proactive pour l'exécution de SBAs

Cette partie décrit la deuxième contribution de nos travaux.

- **Chapter 5** discute du problème de déterminer le meilleur moment pour commencer adaptation proactive avant des défaillances se produisent réellement. Une approche de prédiction en deux phases est introduit pour pouvoir prévoir une défaillance dans le future le plus tôt possible. Cette approche est évaluée par une série de simulations réalistes en utilisant des workflow des types différents.

Conclusions and Perspectives

résume nos contributions et présente des perspectives de recherche pour le futur.

Bibliography

Dans cette partie, une liste des sources utilisées dans cette thèse sont fournis.

Appendix

À la fin, Les annexes fournissent des informations supplémentaires (exemples et les codes sources) pour soutenir cette thèse.

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Un middleware inspiré par la chimie pour l'exécution et l'adaptation flexible d'applications basées sur des services

Nom Prénom de l'auteur : WANG CHEN

Membres du jury :
- Monsieur GIORDANO Maurizio
- Monsieur NEMETH Zsolt
- Monsieur PAZAT Jean-Louis
- Monsieur TEDESCHI Cédric
- Monsieur PEREZ Christian
- Monsieur CÉRIN Christophe

Président du jury : Christophe Cérin

Date de la soutenance : 28 Mai 2013

Reproduction de la these soutenue

☑ Thèse pouvant être reproduite en l'état
☐ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 28 Mai 2013

Signature du président de jury

C. CERIN

Le Directeur,

M'hamed DRISSI

Résumé

Les Architectures Orientées Services (SOA) sont adoptées aujourd'hui par de nombreuses entreprises car elles représentent une solution flexible pour la construction d'applications distribuées. Une Application Basée sur des Services (SBA) peut se définir comme un workflow qui coordonne de manière dynamique l'exécution distribuée d'un ensemble de services. Les services peuvent être sélectionnés et intégrés en temps réel en fonction de leur Qualité de Service (QoS), et la composition de services peut être dynamiquement modifiée pour réagir à des défaillances imprévues pendant l'exécution. Les besoins des architectures orientées services présentent des similarités avec la nature: dynamicité, évolutivité, auto-adaptabilité, etc. Ainsi, il n'est pas surprenant que les métaphores inspirées par la nature soient considérées comme des approches appropriées pour la modélisation de tels systèmes. Nous allons plus loin en utilisant le paradigme de programmation chimique comme base de construction d'un middleware. Dans cette thèse, nous présentons un middleware "chimique" pour l'exécution dynamique et adaptative de SBA. La sélection, l'intégration, la coordination et l'adaptation de services sont modélisées comme une série de réactions chimiques. Tout d'abord, l'instantiation de workflow est exprimée par une série de réactions qui peuvent être effectuées de manière parallèle, distribuée et autonome. Ensuite, nous avons mis en oeuvre trois modèles de coordination pour exécuter une composition de service. Nous montrons que les trois modèles peuvent réagir aux défaillances de type panne franche. Enfin, nous avons évalué et comparé ces modèles au niveau d'efficacité et complexité sur deux workflows. Nous montrons ainsi dans cette thèse que le paradigme chimique possède les qualités nécessaires à l'introduction de la dynamicité et de l'adaptabilité dans la programmation basée sur les services.

Mots Clés : SBA, orchestration, chorégraphie, calcul chimique, instantiation de workflow, coordination de service, adaptation proactive, prédiction de défaillance.

Abstract

With the advent of cloud computing and Software-as-a-Service, Service-Based Application (SBA) represents a new paradigm to build rapid, low-cost, interoperable and evolvable distributed applications. A new application is created by defining a workflow that coordinates a set of third-party Web services accessible over the Internet. In such distributed and loose coupling environment, the execution of SBA requires a high degree of flexibility. For example, suitable constituent services can be selected and integrated at runtime based on their Quality of Service (QoS); furthermore, the composition of service is required to be dynamically modified in response to unexpected runtime failures. In this context, the main objective of this dissertation is to design, to develop and to evaluate a service middleware for flexible execution of SBA by using chemical programming model. Using chemical metaphor, the service-based systems are modeled as distributed, self-organized and self-adaptive biochemical systems. Service discovery, selection, coordination and adaptation are expressed as a series of pervasive chemical reactions in the middleware, which are performed in a distributed, concurrent and autonomous way. Additionally, on the way to build flexible service based systems, we do not restrict our research only in investigating chemical-based solutions. In this context, the second objective of this thesis is to find out generic solutions, such as models and algorithms, to respond to some of the most challenging problems in flexible execution of SBAs. I have proposed a two-phase online prediction approach that is able to accurately make decisions to proactively execute adaptation plan before the failures actually occur.

Keywords: SBA, orchestration, choreography, chemical programming, workflow instantiation, service coordination, (proactive) adaptation, prediction.

N° d'ordre : **13ISAR 15 / D13 - 15**

Institut National des Sciences Appliquées de Rennes

20, Avenue des Buttes de Coëmes • CS 70839 • F-35708 Rennes Cedex

Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96