Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Specialistica in Informatica

JOLIE:
a Service-oriented Programming Language

Tesi di Laurea
in
Linguaggi di Programmazione

Relatore:
Chiar.mo Prof. Gianluigi Zavattaro

Laureando:
Fabrizio Montesi

Correlatore:
Dott. Ing. Claudio Guidi

Sessione I
Anno Accademico 2009/2010

Sommario

Il Service-Oriented Computing (SOC) è un paradigma per la programmazione di applicazioni distribuite basato sulla composizione di servizi. I servizi sono entità computazionali autonome che possono essere descritte, pubblicate e dinamicamente reperite allo scopo di costruire funzionalità sempre più complesse. Al giorno d'oggi la tecnologia di riferimento per il SOC è Web Services, un insieme di specifiche aperte che si concentra su interoperabilità e compatibilità delle infrastrutture software. Questi risultati sono ottenuti, principalmente, attraverso l'adozione del formato XML e del protocollo HTTP come strato sottostante per le comunicazioni.

Uno degli aspetti più importanti del SOC è la *composizione*. Le interfacce pubbliche esposte da ogni servizio permettono la composizione di questi ultimi in *workflow* (flussi di lavoro) complessi, in modo da implementare funzionalità che riutilizzino quelle già offerte dai singoli servizi. La composizione di servizi viene attualmente effettuata tramite due approcci differenti: l'*orchestrazione* e la *coreografia*.

Nell'orchestrazione un singolo servizio, chiamato orchestratore, è responsabile per la composizione ed il coordinamento degli altri servizi al fine di completare un dato compito. La coreografia, invece, descrive le interazioni tra i vari servizi, che eseguono una strategia globale per raggiungere un obiettivo senza un punto di controllo centralizzato. Per queste ragioni viene detto che l'orchestrazione offre un punto di vista locale, mentre la coreografia globale. Al momento i linguaggi di riferimento per l'orchestrazione e la coreografia sono, rispettivamente, WS-BPEL e WS-CDL. WS-BPEL e

WS-CDL non offrono definizioni formali del loro comportamento. Una seria conseguenza di questo è il fatto che la semantica di BPEL può risultare ambigua in alcuni casi: ambienti di esecuzione per BPEL diversi possono dare esito ad esecuzioni diverse dello stesso programma.

In questa tesi viene presentato il linguaggio JOLIE. JOLIE è un progetto open source, pubblicamente reperibile per uso e consultazione. JOLIE è un linguaggio di programmazione orientato ai servizi, attraverso il quale possono essere programmati sia semplici servizi che complessi orchestratori, supportando un notevole grado di scalabilità. JOLIE è stato sviluppando seguendo delle specifiche formali, offerte dal calcolo di processi SOCK, così da permettere di ragionare formalmente su programmi sviluppati con esso.

Ogni programma JOLIE definisce un servizio ed i servizi possono essere facilmente composti in modo da formarne di più complessi. Uno dei più evidenti vantaggi offerti dal linguaggio è rappresentato proprio dalle sue primitive per la composizione. I meccanismi offerti di *aggregation*, *embedding* e *redirection* permettono l'implementazione di tre differenti *patterns* di composizione il cui comune denominatore è il fatto che la loro applicazione restituisce sempre un servizio. Questo avvicina i concetti di servizio e di architettura di servizi, introducendo la possibilità di creare gerarchie (servizi contenenti altri servizi).

Un altro aspetto importante del linguaggio JOLIE è la sua elegante separazione tra il *behaviour* (comportamento) di un servizio e le sue informazioni di *deployment*. Lo stesso behaviour può essere riutilizzato con diversi mezzi e protocolli di comunicazione, ed i collegamenti verso altri servizi possono essere cambiati dinamicamente. JOLIE supporta l'introduzione di nuovi protocolli e mezzi di comunicazione attraverso lo sviluppo di semplici librerie Java, chiamate JOLIE *extensions*. La capacità di poter estendere le possibilità di comunicazione si è dimostrata un fattore chiave nell'integrazione di JOLIE con un ampio spettro di tecnologie. Grazie a questo JOLIE può essere usato per creare applicazioni orientate ai servizi attraverso l'orchestrazione di applicazioni *legacy* non basate, ad esempio, sulle specifiche Web Servi-

ces. Una importante conseguenza di questo punto è la possibilità di creare programmi JOLIE che agiscano come *web server* per applicazioni Web 2.0.

Sommario dei capitoli

Nel **capitolo 1** vengono presentati i concetti chiave del service-oriented computing, con riferimento particolare alle specifiche Web Services. Le principali specifiche Web Services vengono riportate, seguite da uno studio dei concetti più importanti concetti alla base del paradigma. Le definizioni esposte di *service behaviour* (flusso delle attività) e *service engine* (esecuzione delle sessioni) vengono utilizzate alla fine del capitolo per dare la definizione cardine di servizio.

Nel **capitolo 2** viene riportato il calcolo di processi SOCK, la teoria di riferimento per la semantica del linguaggio JOLIE. La sintassi e la semantica del calcolo vengono presentate. In particolare la semantica – strutturata sui tre livelli di *service behaviour*, *service engine* e *service network* – possiede una intuitiva relazione con i concetti esposti nel primo capitolo.

Nel **capitolo 3** vengono presentati i costrutti base del linguaggio JOLIE. Le principali primitive di comunicazione e di composizione delle attività vengono presentate, assieme alla potente sintassi per la manipolazione di dati strutturati.

Nel **capitolo 4** vengono riportati aspetti più avanzati del linguaggio, particolarmente legati alle architetture di sistemi orientati ai servizi. In particolare i costrutti di *aggregation*, *embedding* e *redirection* vengono esposti, in ambito statico e dinamico.

Nel **capitolo 5** vengono presentati i costrutti linguistici e la loro semantica relativamente ai meccanismi per la gestione di errori offerti da JOLIE. Il capitolo offre prima un raffinamento di sintassi e semantica del calcolo di processi SOCK, per poi mostrare i risultati lì ottenuti siano stati trasposti nel linguaggio JOLIE.

Nel **capitolo 6** viene offerta una descrizione dell'implementazione del linguaggio. Le componenti principali dell'implementazione – *Parser*, *Runtime Environment*, *OOIT* e *Communication Core* – vengono riportate e descritte.

Nel **capitolo 7** vengono introdotte alcune importanti tecniche di programmazione orientate ai servizi che sono state sperimentate attraverso il linguaggio JOLIE. Le tecniche fanno particolare riferimento ai costrutti di *aggregation*, *embedding* e *redirection*, e vengono corredate di esempi.

Introduction

Service-Oriented Computing (SOC) is a paradigm for programming distributed applications by means of the composition of services. Services are autonomous, self-descriptive computational entities that can be dynamically discovered and composed in order to build more complex functionalities. As of today the most prominent technology based on SOC is Web Services [49], a set of open specifications that focuses on interoperability and compatibility with existing infrastructures. This is mainly obtained, respectively, through the adoption of the XML [45] document format and by using HTTP [46] as the underlying transport protocol for communications.

One of the most important aspects in SOC is *composition*. The public interfaces exposed by each service allow for the composition of the latter in complex workflows, in order to implement functionalities that reuse those that are already offered by the single services. At the present service composition can be modelled following two different approaches: *orchestration* and *choreography*.

In orchestration a single service, called orchestrator, is responsible for composing and coordinating the other services in order to complete the desired task. Choreography, instead, describes the interactions between the various services, which execute a global strategy in order to achieve the desired result without a single point of control. For these reasons it is said that orchestration offers a local viewpoint whereas choreography offers a global viewpoint. At the present the most credited language for dealing with service orchestration is WS-BPEL [37] (BPEL for short). On the other hand,

the reference language for choreography is WS-CDL [50]. However both languages lack formal foundations, thus undermining the applicability of formal reasoning on programs produced with them. A serious consequence of this fact is that BPEL semantics can be ambiguous in some cases; indeed, different engines for the BPEL language could lead to different executions, as noted in [30].

In the recent years SOC has been, and continues to be, the target of a rising interest both from the Industry and the Academy. In particular, the Industry focused on interoperability through the establishment of standard specifications (such as WS-Addressing [48], WS-Coordination [38] and WS-Security [39]) and integration with existing technologies in order to favor adoption. Academia, on the other hand, has contributed greatly to the understanding of service-oriented systems by means of formal models, where concurrency theory usually plays an important role. These foundational studies are critical in order to perform precise analyses of service-oriented systems, but they usually are very abstract and finding connections between them and real programming languages for service-oriented programming (such as BPEL) can be nontrivial. Academic research is currently following two directions. The former one is to perform formal studies in order to enhance already existing technologies. For instance, various attempts have been made for defining precise semantics for BPEL [31, 40]. This approach leverages the existing user base for the target technology, leading to a potentially higher adoption and feedback from users. However, the whole set of features offered by BPEL is pretty extensive and many works that follow this approach present only a subset of the actual BPEL primitives. The latter direction follows the opposite trail, by starting a formal model and build, from scratch, a new technology based upon its semantics. The advantage of this second option lies in the potential for a clearer separation of concepts and a more solid framework.

In this thesis the JOLIE programming language is presented. JOLIE is an open source project [14], publicly available for consultation and use [24]. To

the best of the author's knowledge, JOLIE is the first full-fledged programming language based upon the service-oriented programming paradigm. By means of JOLIE it is possible to implement both simple services and complex orchestrators, scaling from handling a few clients to very high numbers of connections with invokers and composed services.

JOLIE has been developed by following the aforementioned second path. Indeed, its base semantics follows the formal specifications of the SOCK [20, 17] process calculus. The SOCK calculus was born as a general model for designing service-oriented systems, and has been developed by taking inspiration from the constructs present in renowned process calculi such as CCS [32] and well established technologies such as Web Services. One of the most contradistinctive elements of SOCK is its extensive set of primitives. This is different from the majority of process calculi, where minimality is of critical importance. SOCK, however, strives to maintain a balance between minimality and the level of comprehensiveness that is needed to model real services in a faithful manner. In turn, this makes SOCK particularly suitable to be a base for the implementation of a general language for service-oriented programming such as JOLIE. The fact that JOLIE is based upon a process calculus has been fundamental, e.g., in the development of a solid theory and implementation for error handling [18, 19, 33], which are exposed in Chapter 5.

JOLIE offers a programmer-friendly syntax, resembling those of C and Java, that allows for the fast prototyping of services and their subsequent step-by-step incremental refinement. This is in contrast with the XML-based syntax provided by BPEL which, because of its complexity, is often handled by means of graphical tools. Particular effort has been put into making structured data handling powerful and intuitive. This is because in service-oriented systems it is often the case that one has to handle structured documents (such as XML ones).

Every JOLIE program defines a service, and services can be easily composed in order to form even more complex ones. Indeed, among the most

prominent advantages of JOLIE are its powerful primitives for service composition. The offered mechanisms of *aggregation*, *embedding* and *redirection* allow for the implementation of three different composition patterns whose common denominator is the fact that they always yield another service. This blurs the difference between the concepts of service and service-oriented architecture, introducing the possibilities of creating hierarchies (services containing other services) and building seamless bridges between services that use different interaction protocols or data encodings.

Another important aspect of JOLIE is the elegant separation between the service behaviour and its deployment information. The same behaviour can be used with different communication mediums and protocols. Bindings toward other services can change dynamically, including their communication medium and protocol specifications. Support for new communication means can be added by developing simple Java libraries, called JOLIE extensions. The ability to extend its communication capabilities has proven to be a key factor in integrating JOLIE with a wide range of existing technologies. Thanks to this JOLIE can be used to create service-oriented applications even by orchestrating legacy applications that do not support the Web Services specifications. An important consequence of this point is the possibility to create a JOLIE program to act as a server for Web 2.0 applications.

Structure of the thesis

This thesis is structured as follows:

- Chapter 1 presents the key concepts of service-oriented computing, with references to the Web Services specifications;
- Chapter 2 reports the SOCK process calculus, the theoretical framework that defines the base semantics of JOLIE;
- Chapter 3 presents the basic language constructs of JOLIE;
- Chapter 4 presents more advanced features of the language, particularly related to the architecture of a service-oriented system;
- Chapter 5 reports the constructs and semantics of the error handling mechanisms of JOLIE;
- Chapter 6 describes the implementation of the language;
- Chapter 7 introduces some important service-oriented programming techniques that have been experimented with JOLIE.

Finally, at the end, conclusions and future work are reported.

Contents

Chapter 1

Background and key concepts

Service-Oriented Computing (SOC) is a paradigm for programming distributed applications by means of the composition of services. Services are autonomous, self-descriptive computational entities that can be dynamically discovered and composed in order to build more complex functionalities.

This chapter offers an overview of SOC in its most renowned definition, developed inside the scope of the Web Services technology. Afterwards, a more conceptual description of the service-oriented paradigm, derived from foundational studies, is provided. The latter description is relevant w.r.t. this thesis because its definitions have been inspired both by SOCK, the process calculus behind JOLIE, and practical experience with the JOLIE language itself.

1.1 Service-Oriented Computing and Web Services

As of today the most prominent technology based upon SOC is Web Services, a set of open specifications that focuses on interoperability and compatibility with existing infrastructures. This is mainly obtained, respectively, through the adoption of the XML [45] document format and by using HTTP [46] as the underlying transport protocol for communications.

The base set of specifications offered by Web Services addresses the problems of *data exchange*, *service description* and *service discovery*. These specifications are briefly presented in the following.

- **SOAP: Simple Object Access Protocol** [47]. This specification defines the data format Web Services must use for reading and writing messages.

- **WSDL: Web Service Description Language** [51]. This specification deals with the description of a Web Service interface. A WSDL document defines how a service may exchange messages with other services. The fundamental concepts of this specification are those of *operation* and *port*. Operations represent the basic communication primitives that services can exploit for exchanging messages. There are four operation types, each one related to a specific communication pattern:

  - One-Way: the service receives a message;
  - Request-Response: the service receives a message, and sends a correlated response message;
  - Notification: the service sends a message;
  - Solicit-Response: the service sends a message, and receives a correlated response message.

  Operations are coupled with *message types* and then grouped into *port types*. Finally, port types are joined with *binding information* so to define a *port*. As such, ports contain all the necessary data for telling how a service can interact with the rest of the distributed system.

- **UDDI: Universal Description Discovery and Integration** [36]. This specification defines a standard interface for *service registries*. A service registry allows for the dynamic discovery of other services in the distributed system: services can call the registry and perform queries so to get binding information for the services they are looking for.

1.1.1 Service composition: orchestration and choreography

One of the most important aspects in SOC is *composition*. Indeed, the fact that services offer a public interface in a distributed system allows for their composition in complex workflows, so to implement functionalities that reuse those that are already offered by the composed services. At the present service composition can be modeled by following two different approaches: *orchestration* and *choreography*.

In orchestration a single service, called orchestrator, is responsible for composing and coordinating the other services in order to complete the desired task. Choreography, instead, deals with the description of the interactions between the various services, which execute a global strategy in order to achieve the desired result without a single point of control. For these reasons it is said that orchestration offers a local viewpoint whereas choreography offers a global viewpoint.

The different viewpoints offered by choreography and orchestration are both useful in the implementation of a service-oriented system. Choreography is better suited for the description of interaction protocols and of complex distributed sessions, whereas orchestration allows the programmer to focus on the implementation of each single service. Moreover, the two approaches allow to choose the most convenient viewpoint when one has to perform the verification of different system properties. Deadlock-freedom of a service system, for instance, is usually checked at the level of choreography whereas performance evaluation of services can be better examined at the level of orchestration. The reference Web Services languages for orchestration and choreography are, respectively, WS-BPEL [37] and WS-CDL [50].

1.2 Key concepts: a definition for service

The Web Services specifications are vast and do not offer formal definitions of the entities involved in a service-oriented architecture. For this

reason various efforts have been made into distilling the key concepts upon which SOC is based. The work performed in this thesis has been relevant in one of these efforts, which is reported in [21]; there, a first attempt at presenting service-oriented computing as a full-fledged programming paradigm is made. The definitions reported in that attempt are useful for a better understanding of the following chapters and, as such, they are also briefly presented in this section.

The central definition that is going to be exposed here is that of *service*, the most important concept of the service-oriented paradigm. The definition of service for the W3C Working Group [5] is:

> *"A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities. To be used, a service must be realized by a concrete provider agent."*

This definition is correct but one could argue that it is too abstract because too many things could be a service. Before giving a more precise definition of service the concepts of service *behaviour*, *engine* and *description* are presented, as they are integral parts of it.

1.2.1 Behaviour

Defining the *behaviour* of a service requires the introduction of two other basic concepts: service *activities* and their *composition* in a workflow. Activities represent the basic functional elements of a behaviour, whereas their composition represents the logical order in which they can be executed. Work-flow composition is a key aspect of the service-oriented programming paradigm, coming from the business process language WS-BPEL. Behaviours are defined as follows:

> *The behaviour of a service is the definition of the service activities composed in a workflow.*

Service activities are grouped in three categories:

- *communication activities*: they deal with message exchanges between services;
- *functional activities*: they deal with data manipulation;
- *fault activities*: they deal with faults and error recovery.

Communication activities

Communication activities are called *operations*, as in WSDL. Operations are divided into input operations and output operations. The former provide a means for receiving messages from an external service where the latter are used for sending messages. Their definitions resemble those already seen for WSDL:

- Input operations
  - **One-Way**: it is devoted to receive a message.
  - **Request-Response** : it is devoted to receive a request message and to send a response message back to the invoker.
- Output operations
  - **Notification**: it is devoted to send a message.
  - **Solicit-Response**: it is devoted to send a request message and to receive a response message from the invoked service.

Output operations require the specification of a target endpoint to which the message has to be sent. At the level of behaviour such an endpoint abstractly refers to a service. Such service is referred to as *receiving service*.

Functional activities

Functional activities allow for the manipulation of internal data by providing all the basic operators for expressing computable functions.

Fault activities

Fault activities are devoted to the management of faults and are briefly reported in the following list.

- **Fault raising**: it deals with the signalling of a fault.
- **Fault handler**: it defines the activities to be performed when a fault must be handled.
- **Termination handler**: it defines the activities to be performed when an executing activity must be terminated before its ending.
- **Compensation handler**: it defines the activities to be performed for reverting a successfully finished activity.

1.2.2 Engine

An engine is a machinery able to create, execute and manage service sessions. The concept of session is crucial to service-oriented programming and must be addressed before giving a definition of service engine.

Session

The definition of session follows:

> *A service session is an executing instance of a service behaviour equipped with its own local state.*

A key element of the service-oriented programming paradigm is session identification. In general a session is identified by a part of its own local state, which can be programmatically defined by means of *correlation sets*. Correlation sets is a mechanism provided by WS-BPEL; it has been formalized in SOCK, COWS [29] and [44]. In order to explain this mechanism, a simple notation is introduced in the following. A session is represented by a couple (P, S); P represents a behaviour in a given formalism and S represents the local state. States are modelled functions from variables to names,

$S : Var \rightarrow Values$, where Var is the set of variables and $Values$ the set of values[1]. Now, let us consider two sessions with same behaviour but different local states S_1 and S_2:

$$s_1 := (P, S_1) \quad s_2 := (P, S_2)$$

s_1 is said to be not distinguishable from s_2 if $S_1 = S_2$[2]. Now, let us consider both $\mathcal{S}_1$ and $\mathcal{S}_2$ as a composition of states defined on disjoint domains:

$$\mathcal{S}_1 = S_{11} \oplus S_{12} \qquad \mathcal{S}_2 = S_{21} \oplus S_{22}$$

where $\oplus$ is a composition operator over states[3]. If S_{11} and S_{21} are the correlation sets, respectively, for s_1 and s_2 then the two sessions are said to be not distinguishable by correlation iff $S_{11} = S_{21}$.

Session management

Session management involves all the actions performed by a service engine in order to create and handle sessions. In order to achieve this task, a service engine provides the following functionalities:

- **Session creation**. Sessions can be created in two different ways:
  - when an external message is received on a particular operation of the behaviour. Some operations can be marked as session initiators. When a message is received on a session initiator operation, a session can be started.
  - when a user manually starts it. A user can launch a service engine which immediately executes a session without waiting for an external message. Such a session is denoted as *firing session*.

[1]For the sake of brevity, both states and messages are represented as flat mappings from variables to values. The introduction of structured and typed values does not alter the insights presented here.

[2]Let Σ be the set of states, then equality for states is defined as $=: (\Sigma \times \Sigma)$ where $S_1 = S_2$ if $D = Dom(S_1) = Dom(S_2) \wedge \forall x \in Dom(S_1)\, S_1(x) = S_2(x)$

[3]$\oplus : \Sigma \times \Sigma \rightarrow \Sigma$ where $S_1 \oplus S_2(x) = S_1(x)\ if\ x \in Dom(S_1), S_2(x)\ if\ x \in Dom(S_2) \wedge x \notin Dom(S_1), undefined\ otherwise$

- **State support**. The service engine also provides the support for accessing data which does not reside into a session local state: the *global state* and the *storage state*. Summarizing, it is possible to distinguish three different kind of data resources, here called *states*, that can be accessed and modified by a session:
  - a **local state**, which is private and not visible to other sessions. This state is deleted when the session finishes;
  - a **global state** which is shared among all the running sessions. This state is deleted when the engine stops;
  - a **persistent state** which is shared among all the running sessions and whose persistence is independent from the execution of a service engine (e.g. a database or a file).
- **Message routing.** Since a session is identified by its correlation set, the engine must provide the mechanisms for routing the incoming messages to the right session. The session identification issue is raised every time a message is received. For the sake of generality, it is not possible to assume that some underlying application protocol such as WS-Addressing [48] or other transport protocol identification mechanisms such as HTTP cookies are always used for identifying sessions. Correlation sets can be used as a generalization of the various mechanisms used for routing incoming messages. Its functioning can be summarized as follows. A message M can be seen as a function from variables to values: $M \in \Sigma$. Similarly to states, it is possible to define correlation sets for messages. Let us consider $M = M_1 \oplus M_2$, where M_1 is the correlation set for message M, and a correlation function $c : Var \rightarrow Var$ which maps message variables to state variables. Then, a message M must be routed to the session s whose state is $S = S_1 \oplus S_2$, where S_1 is the correlation set, if:

$$\forall x \in Dom(M_1), c(x) \in Dom(S_1) \Rightarrow$$
$$S(c(x)) = M(x) \vee S(c(x)) \; is \; undefined$$

Informally, a message can be routed to a session only if its correlated data corresponds to that of the session. The correlation function, c, is the concrete means used by programmers for defining correlation. For each incoming message it is possible to define a specific correlation function and the correlation set, which identifies the session, is indirectly defined by the union of the codomains of all the defined correlation functions. In the case that the correlation set is not correctly programmed more than one running session could be correlated to an incoming message, causing the session which has to receive the message to be nondeterministically selected.

- **Session execution**. Session execution deals with the actual execution of a created session behaviour equipped with all the required state supports. Sessions can be executed sequentially or concurrently. The majority of existing technologies share the idea that sessions are to be executed concurrently, but the sequential case allows for the controlling of some specific hardware resource which needs to be accessed sequentially. As an example, consider a cash withdrawal machine which starts sessions sequentially due to its hardware nature. Such an aspect can be very important from an architectural point of view, because it can raise system deadlock issues if not considered properly, as it has been shown in [16].

Engine definition

The definition of service engine can now be presented, leveraging the previous ones:

An engine is a machinery able to manage service sessions by providing session creation, state support, message routing and session execution capabilities.

1.2.3 Service description

A service description provides all the necessary information for interacting with a service. Service descriptions are composed by two parts: interface and deployment.

Interface

Service interfaces contain abstract information for performing compatibility checks between services, abstracting from low-level details such as communication data protocols and transports. Interfaces are structured on three different levels:

- **Functional.** It reports all the input operations used by the behaviour for receiving messages from other services or applications. An operation description is characterized by a name, and its request and response message types. Tracing a comparison with the Web Services technology, this level is well represented by the WSDL specifications [51]. At this level, only message type checks on the interface are required in order to interact with a service.

- **Workflow.** It describes the workflow of the behaviour. In a workflow, input operations could not be always available to be invoked but they could be enabled by other message exchanges by implementing some of high level application protocol. Thus, it is fundamental to know how a service workflow behaves in order to interact with it correctly. In the Web Services technology this level could be provided by means of an Abstract-BPEL [37] description; WSDL 2.0 specifications [4] provide Message Exchange Patterns (MEP) which allows for the description of custom service interaction patterns.

- **Semantics.** It offers semantic information about the service and the specific functionalities provided by it. It is usually provided by using some kind of ontology with specific languages such as OWL-S [2].

Service interfaces are strictly related to service discovery, which is a key element of the service-oriented programming paradigm. Discovery issues are addressed by search and compatibility check algorithms over interface repositories, also called registries. These algorithms differ depending on which interface type is considered.

Deployment

The deployment part is in charge of binding the service interface with network locations and protocols. A service, e.g., could receive messages by exploiting the HTTP protocol or the SOAP over HTTP one, but the choice is potentially unlimited because new protocols may always be created. This task is achieved by means of port declarations. There are two kinds of ports: *input ports* and *output ports*. The former allow for the declaration of all the input endpoints able to receive messages exhibited by the service engine, whereas the latter bind target location and transport protocol to the *receiving services* of the behaviour. In other words, output ports allow for the concrete connection with the services to invoke. In general, a port can be defined as follows:

> *A port is an endpoint equipped with a network address and a communication protocol joined to an interface whose operations will become able to receive or send messages. Ports which enables operations to receive requests are called input ports. Conversely, ports that enable output communications toward other services are called output ports.*

A service engine needs to be joined with deployment information in order to receive and send messages.

1.2.4 Service definition

The definition of service is based upon all the aforementioned concepts:

A service is a deployed service engine whose sessions animate a given service behaviour.

Chapter 2

Foundations

The JOLIE language is based upon a theoretical framework featuring a process calculus for service orchestration, SOCK [20, 17] (Service-Oriented Computing Kernel), thus enabling for formal reasoning on JOLIE programs. To the best of the author's knowledge, SOCK is the only calculus offering a native primitive for performing Request-Response communications. This is important for two reasons; on the one hand, it allows for a more direct mapping between SOCK and its implementation in JOLIE and, on the other hand, it has proven to play a special role in the study of some behaviours, e.g. error handling. More details about the latter point are given in Chapter 5.

SOCK is a calculus for modeling service-oriented systems, inspired by WSDL and BPEL. Its primitives include both uni-directional (One-Way) and bi-directional (Request-Response) WSDL communication patterns, control primitives from imperative languages, and parallel composition from concurrent languages. SOCK is structured on three layers: (i) the *service behaviour* layer specifies the actions performed by a service, (ii) the *service engine* layer deals with state, service instances and correlation sets, and (iii) the *service network* layer allowing different engines to interact. The three layers are described in detail in the following sections.

2.1 Service behaviour layer

The service behaviour layer describes the actions performed by services. Actions can be operations on the state (SOCK has a state like that of many imperative languages), or communications according to the One-Way and Request-Response communication patterns. Basic actions can be composed using composition operators. In SOCK services are identified by the name of their operations, and by their location. Locations are managed at the service network layer. In order to model those aspects the following (disjoint) sets are used: Var, ranged over by x, y, for variables, Val, ranged over by v, for values, $\mathcal{O}$, ranged over by o, for One-Way operations, and $\mathcal{O}_R$, ranged over by o_r for Request-Response operations. Also, Loc is a subset of Val containing locations, ranged over by l. A corresponding subset of Var, $VarLoc$, contains location variables and is ranged over by z. Finally, vectors are represented through this notation: $\vec{k} = \langle k_0, k_1, ..., k_i \rangle$.

The syntax for service behaviour processes, ranged over by $P, Q, \ldots$, is defined in Table 2.1. SC denotes the set of service behaviour processes. $\mathbf{0}$ is the inactive process. Outputs can be Notifications $\overline{o}@z(\vec{y})$ or Solicit-Responses $\overline{o_r}@z(\vec{y}, \vec{x})$, corresponding, respectively, to the client side of One-Way and Request-Response communication patterns. A Notification operation $\overline{o}@z(\vec{y})$ invokes the operation named o (with $o \in \mathcal{O}$) of a service located at the location stored in location variable z. Also, $\vec{y}$ is a vector of variables containing the values to be communicated during the invocation. Similarly, a Solicit-Response operation $\overline{o_r}@z(\vec{y}, \vec{x})$ invokes using a Request-Response communication pattern operation o_r (now $o_r \in \mathcal{O}_R$, since names of Request-Response operations are different from names of One-Way operations) of a service located at the location stored in location variable z. Again $\vec{y}$ is a vector of variables containing the values to be communicated during the invocation. In addition, now $\vec{x}$ is the vector of variables that will be assigned the values received as answer of the invocation. Dually, inputs can be One-Ways $o(\vec{x})$ or Request-Responses $o_r(\vec{x}, \vec{y}, P)$ where the notations are as above, with $\vec{y}$ containing values to be sent and $\vec{x}$ containing variables that will receive

$$\epsilon ::= o(\vec{x}) \mid o_r(\vec{x}, \vec{y}, P) \qquad \overline{\epsilon} ::= \overline{o}@z(\vec{y}) \mid \overline{o_r}@z(\vec{y}, \vec{x})$$

$P, Q, \ \ldots ::=$	ϵ	*input*
	$\overline{\epsilon}$	*output*
	$x := e$	*assignment*
	$P; Q$	*sequential composition*
	$P \mid Q$	*parallel composition*
	$\sum_{i \in W} \epsilon_i; P_i$	*nondeterministic choice*
	$if\ \chi\ then\ P\ else\ Q$	*deterministic choice*
	$while\ \chi\ do\ (P)$	*iteration*
	$\mathbf{0}$	*null process*
	$o_r(\vec{x})$	*response in solicit*
	$Exec(P, o_r, \vec{y}, l)$	*Request-Response execution*

Table 2.1: Service behaviour syntax

the communicated values. Additionally, P is the process to be executed between the request and the response. Essentially, a Notification $\overline{o}@z(\vec{y})$ will interact with a One-Way $o(\vec{x})$ located at the location stored in z, and values in variables $\vec{y}$ will be sent and copied inside variables in $\vec{x}$. Consider instead a Solicit-Response $\overline{o_r}@z(\vec{y}, \vec{x})$ and a corresponding Request-Response $o_r(\vec{x}_1, \vec{y}_1, P)$. After the invocation values from $\vec{y}$ are copied into $\vec{x}_1$. Then process P is executed on the server side. Finally the answer in $\vec{y}_1$ is sent back to the client and copied into variables $\vec{x}$. Only at this point the execution at the client side can continue.

Assignment $x := e$ assigns the result of the expression e to the variable $x \in Var$ (state is local to each behaviour). The syntax of expressions is not presented: by assumption they include the arithmetic and boolean operators, values in Val and variables. Var is a function that given an expression e computes the set of variables in e, and $[\![e]\!]$ is the evaluation of ground expression e. χ ranges over boolean expressions. $P; Q$ and $P \mid Q$ are, re-

spectively, sequential and parallel composition. $\sum_{i \in I} \epsilon_i; P_i$ is input-guarded non-deterministic choice: whenever one of the input operations ϵ_i (either a One-Way or a Request-Response) is invoked, continuation P_i is executed. Also, *while* χ *do* (P) models iteration.

The two last operators in Table 2.1 are not part of the static syntax, but they are used to give semantics to the calculus. The former, $o_r(\vec{x})$ is used to wait for the response in a solicit-Response interaction. Note that, although it looks similar, it is not a One-Way, since operation o_r is a Request-Response one. The latter, $Exec(P, o_r, \vec{y}, l)$ is a running Request-Response: P is the running process, o_r the operation name, $\vec{y}$ the vector of variables to be used for the answer, and l the client location. The operation name and the client location are needed to send back the answer.

Semantics. The service behaviour layer does not deal with state, leaving this issue to the service engine layer. Instead, it generates all the transitions allowed by the process behaviour, specifying the constraints on the state that have to be satisfied for them to be performed. The state, and the conditions on it, are substitutions of values for variables. σ is used to range over substitutions, and $[\vec{v}/\vec{x}]$ for the substitution assigning values in $\vec{v}$ to variables in $\vec{x}$. Given a substitution σ, $\mathrm{Dom}(\sigma)$ is its domain.

The semantics follows the idea exposed above: the labels contain all the possible actions, together with the necessary requirements on the state. Formally, let *Act* be the set of actions, ranged over by a. In order to simplify the interaction with upper layers, structured labels of the form $\iota(\sigma : \theta)$ are used, where ι is the kind of action while σ and θ are substitutions containing respectively the assumptions on the state that should be satisfied for the action to be performed and the effect on the state.

Definition 2.1.1 (Service behaviour layer semantics)**.** $\rightarrow \subseteq SC \times Act \times SC$ *is the least relation which satisfies the rules of Tables 2.2 and is closed w.r.t. structural congruence* $\equiv$*, the least congruence relation satisfying the axioms in Table 2.3.*

$$\text{(One-WayOut)}\quad \overline{o}@z(\vec{x}) \xrightarrow{\overline{o}(\vec{v})@l(l/z,\vec{v}/\vec{x}:\emptyset)} \mathbf{0} \qquad \text{(Request-Response)}\quad Exec(\mathbf{0}, o_r, \vec{y}, l) \xrightarrow{\downarrow\overline{o_r}(\vec{v})@l(\vec{v}/\vec{y}:\emptyset)} \mathbf{0}$$

$$\text{(One-WayIn)}\quad o(\vec{x}) \xrightarrow{o(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathbf{0} \qquad \text{(Request)}\quad o_r(\vec{x}, \vec{y}, P) \xrightarrow{\uparrow o_r(\vec{v})@l(\emptyset:\vec{v}/\vec{x})} Exec(P, o_r, \vec{y}, l)$$

$$\text{(Solicit)}\quad \overline{o_r}@z(\vec{x}, \vec{y}) \xrightarrow{\uparrow\overline{o_r}(\vec{v})@l(l/z,\vec{v}/\vec{x}:\emptyset)} o_r(\vec{y}) \qquad \text{(Solicit-Response)}\quad o_r(\vec{x}) \xrightarrow{\downarrow o_r(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathbf{0}$$

$$\text{(Request-Exec)}\quad \frac{P \xrightarrow{a} P'}{Exec(P, o_r, \vec{y}, l) \xrightarrow{a} Exec(P', o_r, \vec{y}, l)} \qquad \text{(Assign)}\quad \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(e) \quad [\![e\sigma]\!] = v}{x := e \xrightarrow{\tau(\sigma:v/x)} \mathbf{0}}$$

$$\text{(If-then)}\quad \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = true}{if\ \chi\ then\ P\ else\ Q \xrightarrow{\tau(\sigma:\emptyset)} P} \qquad \text{(Else)}\quad \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = false}{if\ \chi\ then\ P\ else\ Q \xrightarrow{\tau(\sigma:\emptyset)} Q}$$

$$\text{(Iteration)}\quad \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = true}{while\ \chi\ do\ (P) \xrightarrow{\tau(\sigma:\emptyset)} P; while\ \chi\ do\ (P)} \qquad \text{(No-Iteration)}\quad \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = false}{while\ \chi\ do\ (P) \xrightarrow{\tau(\sigma:\emptyset)} \mathbf{0}}$$

$$\text{(Sequence)}\quad \frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q} \qquad \text{(Parallel)}\quad \frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad \text{(Choice)}\quad \frac{\epsilon_i \xrightarrow{a} Q_i \quad i \in I}{\sum_{i\in I} \epsilon_i; P_i \xrightarrow{a} Q_i; P_i}$$

Table 2.2: Rules for service behaviour layer

Rule One-WayOut defines the solicit operation: the first part of the label, $\overline{o}(\vec{v})@l$ is the actual action. Here l is the location of the invoked service, taken from variable z. The other two arguments define the effect and the requirements on the state respectively. Substitution $[l/z, \vec{v}/\vec{x}]$ specifies that this transition can be performed only if the state assigns value l to variable z and values in $\vec{v}$ to variables in $\vec{x}$. This assumption will be checked by the service engine layer. The empty substitution $\emptyset$ specifies that the operation does not affect the state. Rule One-WayIn corresponds to the One-Way operation. Here there are no conditions on the state, but a state update is required by the label: values in $\vec{v}$ should be assigned to variables in $\vec{x}$.

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0}; P \equiv P \qquad \langle \mathbf{0} \rangle \equiv \mathbf{0}$$

Table 2.3: Structural congruence

State updates are performed by the service engine layer. The solicit and the One-Way operations are synchronized in the service network layer.

Similarly rules SOLICIT and REQUEST start a solicit-response operation. The main difference between the solicit-response and the notification is that the solicit-response leaves an operation waiting for the response. The Request-Response instead, after invocation, becomes an active construct executing process P, and storing all the information needed to send back the answer. The execution of P is managed by rule REQUEST-EXEC. When the execution of P is terminated, rule REQUEST-RESPONSE sends back the answer, exploiting the stored information about the name of the operation and the location of the invoker. This synchronizes with rule SOLICIT-RESPONSE on the client side, concluding the communication pattern.

The other rules in Table 2.2 are standard, apart from the fact that the label stores the conditions on the state. For instance assignment $x := e$ produces an internal step, and requires to update the state by assigning value v to variable x, provided that the state provides a substitution σ for variables in e such that the evaluation of $e\sigma$ is v.

2.2 Service engine layer

In a service engine, all the executed sessions of a service behaviour are joined by a state and a correlation set.

The service engine calculus syntax is:

$$I ::= (P, \mathcal{S}) \quad | \quad I \,|\, I \qquad Y ::= c \triangleright P[I]$$

where P is a service behaviour process, $\mathcal{S}$ is a state and c is a correlation set, i.e. a subset of Var.

A state is for us a substitution of values for variables. Given a state $\mathcal{S}$ and a substitution σ we say that $\mathcal{S}$ satisfies σ, written $\mathcal{S} \vdash \sigma$, if σ is a subset of $\mathcal{S}$. We also write $\mathcal{S}(x) = \perp$ when x is undefined in state $\mathcal{S}$.

Semantics. The lts rules for service engine state layer follow.

(Engine-State 1)
$$\frac{P \xrightarrow{\iota(\sigma:\vec{v}/\vec{x})} P', \mathcal{S} \vdash \sigma, \iota \neq \tau}{(P, \mathcal{S}) \xrightarrow{\iota(\vec{v}/\vec{x}:\mathcal{S}(\vec{x}))} (P', \mathcal{S} \uplus [\vec{v}/\vec{x}])}$$

(Engine-State 2)
$$\frac{P \xrightarrow{\iota(\sigma:\emptyset)} P', \mathcal{S} \vdash \sigma}{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S})}$$

(Engine-State 3)
$$\frac{P \xrightarrow{\tau(\sigma:\vec{v}/\vec{x})} P', \mathcal{S} \vdash \sigma}{(P, \mathcal{S}) \xrightarrow{\tau} (P', \mathcal{S} \uplus [\vec{v}/\vec{x}])}$$

(Engine-State 4)
$$\frac{P \xrightarrow{\iota} P', \iota \in \{th(f), inst(\mathcal{H})\}}{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S})}$$

Rule Engine-State 1 verifies that the condition σ on the state is satisfied and updates it with $[\vec{v}/\vec{x}]$. The old values are tracked in the label since they are needed to check correlation of messages. The second rule is simpler, since it deals with actions that do not update the state and do not require correlation. The third one deals with assignments. Rule Engine-State 4 treats faults or compensation installations that reach service engine.

When an input is received by a service engine, it is possible that several sessions are waiting on the same operation. The session chosen for message delivery depends on the values of the correlated variables. Given two values v and w, a variable x and a correlation set c, v is correlated to x coherently with c, written $v/x \vdash_c w$, if any of the following conditions hold:

- the variable x belongs to c and its actual value is $w = v$,
- the variable x belongs to c and $w = \perp$,
- the variable x does not belong to c.

Rules for service engine correlation lts layer follow.

(Correlated)
$$\frac{I \xrightarrow{\iota(\vec{v}/\vec{x}:\vec{w})} I', \vec{v}/\vec{x} \vdash_c \vec{w}}{I \xrightarrow{\iota,c} I'}$$

(NotCorrelated)
$$\frac{I \xrightarrow{\iota} I'}{I \xrightarrow{\iota,c} I'}$$

(Par)
$$\frac{I \xrightarrow{\iota,c} I'}{I|I'' \xrightarrow{\iota,c} I'|I''}$$

The first rule ensures that an input is received by a correlated session, while the second one deals with actions that need no correlation. In this case any correlation set is fine. The last rule deals with parallel composition.

The following defines rules for session execution and creation:

$$\text{(Exec)}\quad \frac{I \xrightarrow{\iota,c} I'}{c \triangleright P[I] \xrightarrow{\iota} c \triangleright P[I']}$$

$$\text{(Spawn)}\quad \frac{(P, \mathcal{S}_\perp) \xrightarrow{\iota,c} (P', \mathcal{S}),\ \nexists \mathcal{S}_i \in \text{extr}(I).(P, \mathcal{S}_i) \xrightarrow{\iota,c} (P', \mathcal{S}'_i), \iota \in In}{c \triangleright P[I] \xrightarrow{\iota} c \triangleright P[I|(P', \mathcal{S})]}$$

In rule Spawn $\mathcal{S}_\perp$ is the state undefined on all variables and $\text{extr}(I)$ is a function extracting all states occurring in I. The first rule allows to execute an existing session, while the second spawns a new session provided that an input that cannot be handled by the available sessions is received.

A service engine may be equipped with additional information, such as a directive for imposing the sequential execution of sessions. The reader interested in the complete semantics of SOCK for service engines is referred to [16, 17].

2.3 Service network layer

The service network layer allows for the composition of different engines into a system. The engines are composed in parallel and equipped with a location that allows for their unambiguous identification. The syntax is:

$$E ::= Y@l \ \mid \ E \parallel E$$

A service engine system E can be a located service engine $Y@l$ or a parallel composition of them. The semantics is defined by the rules in Table 2.4 and closed w.r.t. the structural congruence $\equiv$ therein.

Rule Lift propagates an action to a located engine. Rule NormalSync allows to synchronize an output with the corresponding input (according

$$\frac{Y \xrightarrow{\iota} Y'}{Y@l \xrightarrow{\iota} Y'@l}\ (\textsc{Lift}) \qquad \frac{Y@l' \xrightarrow{\lambda@l} Y'@l' \quad Z@l \xrightarrow{\lambda'} Z'@l \quad compl(\lambda,\lambda')}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}\ (\textsc{NormalSync})$$

$$\frac{E_1 \xrightarrow{\iota} E_1'}{E_1 \parallel E_2 \xrightarrow{\iota} E_1' \parallel E_2}\ (\textsc{Par-Ext}) \qquad \frac{Y@l' \xrightarrow{\uparrow\overline{o_r}(\vec{v})@l} Y'@l' \quad Z@l \xrightarrow{\uparrow o_r(\vec{v})@l'} Z'@l}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}\ (\textsc{Solicit-RequestSync})$$

$$\frac{E_1 \equiv E_1' \ ,\ E_1' \xrightarrow{\gamma} E_2',\ E_2' \equiv E_2}{E_1 \xrightarrow{\gamma} E_2}\ (\textsc{CongrE})$$

where $compl(\overline{o}(v), o(v)), compl(\downarrow \overline{o_r}(v), \downarrow o_r(v)), compl(\overline{o_r}(f), o_r(f))$.

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

Table 2.4: Rules for service network layer

to the predicate *compl*), checking that the location of the receiving process is the desired one. Rule Solicit-RequestSync additionally checks the correctness of the guess in the input label about the location of the invoking process. The location is needed only for Request-Response, since it is used by the server to send back the answer. Finally rule Par-Ext deals with parallel composition.

Chapter 3

Language basics

This chapter is devoted to the exposition of the basic language constructs of JOLIE.

A JOLIE program is composed by two parts, a *behavioural* part and a *deployment* one. The behavioural part contains the workflow definition of the orchestrator, whereas the deployment part contains directives for the execution engine and specifies information for the integration of the orchestrator in a service-oriented architecture. This separation is provided so to allow for the reuse of existing behavioural definitions in different service environments (by changing the deployment information) and for the reuse of deployment information with compatible workflows. The offered syntax maps nicely to the layered structure of SOCK: the service behaviour layer is represented by the behavioural part and the service engine layer by the deployment part. As the service network layer is only a semantic layer, there is no corresponding JOLIE syntax for it. Instead, this layer is implemented by a part of the interpreter (specifically the Communication Core, which is responsible for performing communications, see Chapter 6).

As far as the behavioural language is concerned, it is possible to interact with other services by means of communication primitives inspired by WSDL operations (One-Way, Request-Response, Notification and Solicit-Response), to synchronize internal parallel processes, to use the classic `while`

loop instruction and the `if-then-else` conditional statement. Moreover, the programmer is allowed to compose statements in a workflow by making sequences, parallelisms and non-deterministic choices. Using its communication primitives and its compositional operators, JOLIE can compose other services by exploiting their input operations.

As far as the deployment language is concerned, its grammar structure is composed by two main parts. The first part contains the deployment directives (execution modality, the state mode (persistent or not persistent) and the correlation set of the orchestrator; these directives map the same features provided by the service engine layer of SOCK for dealing with sessions and service statefulness. The second part deals with interfaces and contains all the information needed for interaction with other services: operations, port types, data protocols and communication endpoints.

The structure of a JOLIE program is given by the following syntax:

Program ::= **Deployment Behaviour**

The definitions of nonterminals **Deployment** and **Behaviour** are exposed in the following sections. For the sake of clarity, the exposition follows a step-by-step augmentation of the definitions of the nonterminals.

Comments are processed before code execution and can be introduced in JOLIE code by means of the same syntax currently in use in the Java language:

```
// This is an inline comment
/*
    This is a multiline
    comment
*/
```

Another feature that is processed before code execution is constant definition. Constants can be defined within the `constants` block, to be introduced in the program preamble, e.g.:

```
constants {
```

```
    MyFirstConstant = 2,
    MySecondConstant = "Hello, world!"
}
```

3.1 Basic behavioural constructs

A JOLIE program must define a `main` procedure, which represents its entry point for execution. The `main` procedure may contain any kind of **Process**. Moreover, it can be preceded or succeeded by definitions of auxiliary procedures and initialization code (whose meaning will be exposed afterwards in, respectively, 3.2.2 and 3.5.1). In the following the Kleene star * is used to indicate zero or more repetitions.

Behaviour ::= **BehaviouralBlock*** `main` { **Process** }
BehaviouralBlock*

BehaviouralBlock ::= `define` *id* { **Process** } *Definition*
| `init` { **Process** } *Initialization code*

id represents an identifier, following the same rules for identifiers given by the Java language.

Processes define the activities to be performed by the service. The most basic process is the no-op one, which simply does nothing:

Process ::= `nullProcess`

3.1.1 Communication statements

The most important behavioural statements are those for performing communications. JOLIE features two communication patterns, inspired by WSDL and formalized in SOCK:

- One-Way: the endpoint receives a message;
- Request-Response: the endpoint receives a message, and sends a response back to the caller.

The two communication patterns can be implemented by using four statements, which follow the semantic rules seen in Chapter 2:

Process	::=	...	
	|	**InputStatement**	
	|	**OutputStatement**	
InputStatement	::=	`op(x)`	*One-Way*
	|	`op(x)(y)` { **Process** }	*Request-Response*
OutputStatement	::=	`op@OPort(x)`	*Notification*
	|	`op@OPort(x)(y)`	*Solicit-Response*

Statement *One-Way* is used to receive a message for operation `op` in variable `x`. Statement *Request-Response* is used to receive a message for operation `op` in variable `x`, execute a **Process** and then send back a response to the caller containing the value of variable `y`. *Notification* and *Solicit-Response* are the dual of the former ones, to be used, respectively, for sending a message to a *One-Way* statement or to a *Request-Response* one. These output statements make use of an *output port name* in order to refer to the binding information necessary for communicating with the desired peer. Output ports will be explained later on in 3.3.2.

Communication statements can make use of variable paths and expressions. These concepts, along with an updated syntax for the statements of interest, are going to be exposed in 3.2.1.

3.1.2 Process composition

Processes can be composed in sequences, parallels and (input guarded) nondeterministic choices, as seen in SOCK. The syntax for sequential and parallel compositions follows:

Process	::=	...	
	|	**Process** ; **Process**	*Sequence*
	|	**Process** | **Process**	*Parallel*

where sequential composition has higher priority (i.e. binds more tightly).

Input guarded nondeterministic choices can be constructed with the fol-

lowing syntax:

Process ::= ...
| **NDInputChoice***
NDInputChoice ::= [**RecvStatement**] { **Process** }

For instance, an implementation of a nondeterministic could look like this:

```
[ logMessage( message ) ] {
    log@InternalLogger( message )
}
[ doNothing() ] {
    nullProcess
}
```

The semantics is the same as in SOCK: when one of the input statements in the choice receives a message, the associated process is executed and the other possible branches are deactivated.

3.1.3 Internal synchronization links

Parallel processes can synchronize by means of synchronization links:

Process ::= ...
| `linkIn ( id )` *Link in*
| `linkOut ( id )` *Link out*

`linkIn` is a blocking statement that waits for a signal on the same *id* to be fired by a corresponding `linkOut` statement. `linkIn` is a blocking primitive, whereas `linkOut` is not.

Let us consider the following example, which composes activities `A` and `B`:

```
main
{
    {
        A;
        linkOut( startB )
    }
```

```
    |
    {
        linkIn( startB );
        B
    }
}
```

This program would first execute `A`, then fire a signal for `startB`. The signal would be received by instruction `linkIn( startB )`, and then `B` would finally be executed.

3.2 Handling data

JOLIE variables are implicitly typed: there is no need to declare their type in advance, as in C or Java. Furthermore their types are dynamic: they can change at runtime depending on the values that the program assigns to them. For instance the following code, where we assign data of two different types to the same variable, is valid:

```
x = 3;
x = "Hello, world!"
```

It is possible to make vectors of values using a syntax that resembles those of other famous languages:

```
vector[0] = 32;
vector[1] = "John";
vector[2] = "Smith"
```

As shown in the example, different elements of the same vector can be of different type (in this case, the first element is an integer and the last two are strings).

JOLIE data structures are organized as trees, similarly (but not equivalently) to XML. The dot operator can be used to access subnodes of a specific

variable:

```
person.name = "John Smith";
person.age = 32
```

It is easy to understand how JOLIE data structures work by comparing them to XML trees. For instance, the structure created in the last example would be equivalent to:

```
<person>
    <name>John Smith</name>
    <age>32</age>
</person>
```

Access to variables is generalized by the concept of *variable path*. A variable path points to the position of a node in a JOLIE data tree. Variable paths are defined by the following syntax:

VariablePath ::= *id* **SubPath**
| *id* **[Expression] SubPath**
SubPath ::= **. VariablePath**
| ϵ

where terminal *id* represents a variable identifier token, defined as in Java language, and nonterminal **Expression** represents an expression (it will be defined more precisely later). The value of the expression is used as index for accessing vectors. Note that whenever a vector index is not specified, JOLIE implicitly considers it to be zero.

In order to make repetitive access to a variable tree less tedious a `with` construct is provided:

Process ::= ...
| `with(` **VariablePath** `)` { **Process** }

Inside a `with` block a particular form of variable paths is available, that of *prefixed variable path*. These variable paths start with a dot, and will be prefixed with the variable path specified inside the round parenthesis at the beginning of the `with` construct. Prefixed variable paths can be used

anywhere a variable path is expected, but are valid only inside a `with` block. For instance, the following code is semantically equivalent to that shown in the last example:

```
with( person ) {
    .name = "John Smith";
    .age = 32
}
```

3.2.1 Expressions

The simplest form of expression is a variable path or a value:

Expression ::= **VariablePath**
| *integer* | *double* | *string*

where *integer*, *double* and *string* values are expressed as in the Java language. Expressions can be constructed by using the classical arithmetic operators:

Expression ::= ...
| **VariablePath + VariablePath** *Sum*
| **VariablePath - VariablePath** *Subtraction*
| **VariablePath * VariablePath** *Multiplication*
| **VariablePath / VariablePath** *Division*

where the operator priority is the same as in the Java language. Still similarly to Java, summing strings returns their concatenation. Increment and decrement unary operators, with the same semantics for numbers of C and Java, along with explicitation of priority are provided:

Expression ::= ...
| **(Expression)**
| **++ VariablePath** *Pre-increment*
| **VariablePath ++** *Post-increment*
| **-- VariablePath** *Pre-decrement*
| **VariablePath --** *Post-decrement*

One can also make use of casts in order to convert values to given types and of type check operators for checking the type of a variable.

Expression ::= ...
| `int(`**VariablePath**`)` *Cast to integer*
| `double(`**VariablePath**`)` *Cast to double*
| `string(`**VariablePath**`)` *Cast to string*
| `is_int(` **VariablePath** `)`
| `is_double(`**VariablePath**`)`
| `is_string(`**VariablePath**`)`

Furthermore, a native operator for determining the length of a vector is provided:

Expression ::= ...
| `#` **VariablePath** *Vector length*

Expressions are mostly used in assignments, as shown in the following:

Process ::= ...
| **VariablePath** `=` **Process** *Assignment*

Now that variable paths and expressions have been presented, the final syntax for input and output communications can be exposed:

InputStatement ::= `op(` **VariablePath** `)` *One-Way*
| `op(` **VariablePath** `)`
`(` **Expression** `)`
{ **Process** } *Request-Response*

OutputStatement ::= `op@OPort(` **Expression** `)` *Notification*
| `op@OPort(` **Expression** `)`
`(` **VariablePath** `)` *Solicit-Response*

3.2.2 Flow control constructs

Execution flow can be controlled by means of some classical imperative constructs, too. Their behaviour is governed by the value of some condition; conditions can be written by composing expressions in the usual way:

Condition ::= **Expression**
| `(` **Condition** `)`
| `!` **Condition**
| **Expression Comparator Expression**

Comparator ::= `<` | `<=` | `>` | `>=` |`==` | `!=`

JOLIE supports loop programming through the `for` and `while` constructs:

Process ::= ...
| `for(` **Process** `,` **Condition** `,` **Process** `)`
{ **Process** } *For loop*
| `while(` **Condition** `)` { **Process** } *While loop*

Deterministic choices can be implemented through the classic if-then-else mechanism:

Process ::= ...
| **IfStatement**

IfStatement ::= `if(` **Condition** `)` { **Process** } **ElseIf** *If-then-else*

ElseIf ::= `else` **IfStatement**
| `else` { **Process** }
| ϵ

3.2.3 Dynamic variable paths

Variable paths may be defined by exploiting the evaluation of some expressions: such cases are called *dynamic variable paths*. Expressions can be used only for referencing subnodes, thus the syntax extension for dynamic variable paths is introduced in nonterminal **SubPath**:

SubPath ::= ...
| `.(` **Expression** `)` **VariablePath** *Dynamic sub path*
| `.(` **Expression** `)[` **Expression** `]`
VariablePath *(with vector index)*

In order to clarify the semantics of this particular construct let us see the following example, where we assign to variables `x`, `y` and `z` the value from `person.age`:

```
person.age = 30;
```

```
x = person.("age");
key = "age";
y = person.(key);
z = person.("a" + "ge")
```

Dynamic variable paths are useful for implementing tables. Consider the case in which, for instance, a service offers the possibility to get the population number of some cities. One could use a tree for storing information about the cities, as in the following (for the sake of simplicity, here such information is statically defined):

```
cities.Copenhagen.population = 530902;
cities.Munich.population = 1326807;
cities.Rome.population = 2731996
```

Then, one could expose a Request-Response operation `getCityPopulation` that receives a city name and returns the related population number:

```
getCityPopulation( cityName )( population ) {
    population = cities.(cityName).population
}
```

Another important feature enabled by dynamic variable paths is the possibility to iterate through the subnodes of a given variable path. This is made possible in conjunction with another flow control construct, `foreach`:

Process ::= ...
| `foreach(` **VariablePath** `:` **VariablePath** `)`
`{` **Process** `}` *Foreach loop*

The `foreach` loop iterates through all the subnode names of the second variable path, assigning each name to the first variable path. Recalling the previous example, a foreach loop could be used in order to iterate through all the stored city names:

```
i = 0;
foreach( cityName : cities ) {
```

```
    names[i] = cityName;
    i++
}
```

At the end of the loop, `names` would be a vector containing all the store city names.

When handling tables one may need to remove some subnodes from a tree or to check if a given subnode is present. These issues are addressed, respectively, by the `undef` and `is_defined` commands:

Process ::= ...
| `undef(` **VariablePath** `)`
Expression ::= ...
| `is_defined(` **VariablePath** `)`

3.2.4 Deep copy and aliases

Data structures can be entirely copied from one variable to another in a single step through the *deep copy* operator:

Process ::= ...
| **VariablePath << VariablePath** *Deep copy*

For instance, the following code copies the entire `person` tree into another variable:

```
personCopy << person
```

Moreover, a variable may be an alias for another variable. This is useful in order to *link* some tree to another one without having to perform a copy of the latter. Whenever a variable path is followed, for each subnode containing an alias the latter is resolved and the path resolution continues. The syntax for aliases is:

Process ::= ...
| **VariablePath -> VariablePath** *Variable path alias*

3.3 Basic deployment constructs

In order to communicate with other services one must precede the behavioural definition of a JOLIE program with its associated deployment information. Deployment information can be defined by using various instructions, which can be freely alternated and can include source code from other files:

Deployment ::= **DeploymentInstruction***

DeploymentInstruction ::= **Include** Inclusion

The main deployment instructions available are those for defining *interfaces*, *message types* and *communication ports*.

3.3.1 Interfaces and message types

Interfaces are sets of operations equipped with information about their request and, in the case of Request-Response operations, response types. Therefore, message types are here exposed before interfaces and then reused afterwards in the explanation for their definition.

Message types

Message types[1] are introduced in the deployment part of JOLIE programs:

DeploymentInstruction ::= ...
| `type` *id* `:` **TypeDefinition**

where *id* is the name for referring to the message type afterwards in other parts of the program.

The simplest possible message type definitions are those using a *native type*. Native types do not define any kind of structure. JOLIE currently supports various native types:

NativeType ::= `int` | `double` | `string` | `raw` | `void` | `any`

Native types `raw`, `void` and `any` deserve a clarification:

[1] The first version of the implementation for message types has been developed by E. Ciotti in [13].

- `raw` is meant for the transmission of raw data streams, under the form of byte arrays;
- `void` means that no value may be contained by the variable;
- `any` tells that any native type assumed by the variable will be accepted.

A **TypeDefinition** may simply specify a native type:

TypeDefinition ::= **NativeType**

The most notable feature offered by JOLIE message types is the possibility to define data structures. This is obtained by defining subnodes of a specific type following a tree-like structure:

TypeDefinition	::=	...	
	|	**NativeType**	
		{ **SubTypeList** }	
	|	**NativeType** { `?` }	*Untyped subnodes*
	|	*id*	*Type link*
	|	`undefined`	*Shortcut for* `any:{?}`
SubTypeList	::=	**SubType**	
	|	**SubType SubTypeList**	
SubType	::=	`.` *id* **Cardinality**	
		`:` **TypeDefinition**	
Cardinality	::=	`[` *int* `,` *int* `]`	*Range*
	|	`[` *int* `, *` `]`	*Lower-bound*
	|	`*`	*Shortcut for* `[0,*]`
	|	`?`	*Shortcut for* `[0,1]`
	|	ϵ	

where *Untyped subnodes* specifies that a node may have any kind of subtree. *Type link* may be used to refer to an already defined type (identified by *id*), so to reuse previous definitions. Moreover, **Cardinality** allows for the definition of the number of possible occurrences of a subnode (so to check the length of subnode vectors).

Below some message type examples are reported. The first one defines a `Person` type with three subnodes: name, age and phoneNumber, the latter

having an unbounded number of occurrences. `Person` is then reused in the definition of type `Family`, associated to a cardinality that imposes at least one occurrence of that subnode.

```
type Person:void {
    .name:string
    .age:int
    .phoneNumber*:string
}

type Family:void {
    .address:string
    .person[1,*]:Person
}
```

Interfaces

Interfaces are collections of operation types. Each operation type is composed by an operation name, a request type and, if it is a Request-Response operation, a response type. Request and response types can be defined either by using a native type, keyword `undefined` or the name of a previously defined type. The grammar for interfaces follows:

DeploymentInstruction	::=	...
	|	`interface` *id* { **OperationGroup** }
OperationGroup	::=	`OneWay:` **OneWayList**
	|	`RequestResponse:` **RRList**
OneWayList	::=	**OneWayOp**
	|	**OneWayOp** , **OneWayList**
OneWayOp	::=	*id* (**OpMessageType**)
RRList	::=	**RequestResponseOp**
	|	**RequestResponseOp** , **RRList**

RequestResponseOp ::= *id* (**OpMessageType**)
(**OpMessageType**)
OpMessageType ::= *id* | `undefined` | **NativeType**

Let us consider an extension of example 3.3.1, where the message types definitions are followed by an interface that refers to them:

```
type Person:void {
    .name:string
    .age:int
    .phoneNumber*:string
}

type Family:void {
    .address:string
    .person[1,*]:Person
}

interface PeopleRegistryInterface {
OneWay:
    addNewFamily(Family)
RequestResponse:
    getFamilyByAddress(string)(Family),
    getPersonByName(string)(Person)
}
```

3.3.2 Communication ports

Communication ports define how communications with other services are actually performed. Two kinds of ports are supported:

- input ports: they deal with exposing input operations to other services;

- output ports: they define how to invoke a set of operations of other services.

Intuitively, the two concepts are the counterparts of each other. Consequently, their syntaxes are quite similar. Ports are based upon the three fundamental concepts of *location*, *protocol* and *interface*.

A location expresses the communication medium, along with its configuration parameters, a service uses for exposing its interface (in the case of an input port) or contacting another service (in the case of an output port). Examples of communication mediums are TCP/IP sockets, Unix sockets, Bluetooth communication channels, local memory channels, etc. A protocol defines how data to be sent or received should be, respectively, encoded or decoded following an isomorphism. Examples of protocols are SOAP, SODEP (a binary protocol specifically developed for JOLIE), HTTP forms, etc. Finally, a port must specify the interface that is accessible through it. The syntax for writing ports strictly resembles this composition:

DeploymentInstruction	::=	...
	|	`inputPort` *id* { **PortInstruction*** }
	|	`outputPort` *id* { **PortInstruction*** }
PortInstruction	::=	`Location:` " *URI* "
	|	`Protocol:` *id* **ProtocolConfiguration**
	|	`Interfaces:` **InterfaceList**
InterfaceList	::=	*id*
	|	*id* **,** **InterfaceList**
ProtocolConfiguration	::=	{ **AssignmentList** }
AssignmentList	::=	**AssignmentStatement**
	|	**AssignmentStatement** **;** **AssignmentList**

A location is specified through a URI (Uniform Resource Identifier), which must indicate the communication medium the port has to use and its related parameters, in this form: *medium*:*parameters*. JOLIE currently supports four mediums: `btl2cap` (Bluetooth L2CAP), `localsocket` (Unix local sock-

ets), `rmi` (Java RMI) and `socket` (TCP/IP sockets). Protocols are referred by name, with the possibility of defining some additional configuration for them. This configuration is given by means of assignments, which are to be treated as inside a `with` block (which implicitly points to the configuration tree of the related protocol). Currently supported protocols are HTTP, HTTPS, GWT-RPC [1], SOAP, SODEP [3], SODEPS and XML-RPC [6].

It is possible to use the syntax shown so far to implement working JOLIE programs. Let us consider the following example, where two listings are given. The first one defines a service that offers an operation for performing the summation of some (integer) numbers, whereas the second one is a client designed to invoke the former. The programs are complete: they include both behavioural and deployment information, so they are executable and would function as expected.

Listing 3.1: A service offering an operation for summing numbers

```
type SumRequest:void {
    .number[2,*]:int
}

interface SumInterface {
RequestResponse:
    sum(SumRequest)(int)
}

inputPort SumInput {
Location: "socket://localhost:80/"
Protocol: soap
Interfaces: SumInterface
}

main
{
```

```
    while( 1 ) {
        sum( request )( result ) {
            for( result = 0; i = 0,
                i < #request.number, i++ )
            {
                result = result + request.number[i]
            }
        }
    }
}
```

Listing 3.2: A client requesting the sum of some numbers

```
type SumRequest:void {
    .number[2,*]:int
}

interface SumInterface {
RequestResponse:
    sum(SumRequest)(int)
}

outputPort SumService {
Location: "socket://localhost:80/"
Protocol: soap
Interfaces: SumInterface
}

main
{
    request.number[0] = 3;
    request.number[1] = 5;
```

```
    request.number[2] = 1;
     // response will be 10
    sum@SumService( request )( response )
}
```

3.4 Procedures and inclusions

Code reuse is mostly implemented in service-oriented architectures through modularization in services, which are then composed as needed. However, programmers writing complex service definitions may need some more basic features for reusing code. JOLIE provides two main features for this purpose: procedure definition and source code inclusion.

3.4.1 Procedures

The definition of procedures callable by other code is performed using the aforementioned *Definition* syntactic rule. Procedures can be simply invoked with their name:

Process ::= ...
| *id*

For instance, the following is a possible usage of a procedure definition:

```
define sumProcedure
{
    sum = x + y
}

main
{
    x = 1;
    y = 2;
    sumProcedure
```

```
}
```

Note that, unlike in other major languages, procedures do not possess a local variable state. Details about this matter are described in 3.5.2.

3.4.2 Source code inclusion

Source code inclusion is a mechanic similar to the `#include` directive of the C language preprocessor, through which one can include the content of another file. JOLIE interprets a file inclusion by substituting the inclusion statement with the content of the file. File inclusions can be freely used both in the behavioural and deployment parts. The syntax for inclusions follow:

Include ::= `include "` *string* `"`

where *string* must be a filepath identifying the file to be included.

If the provided filepath is absolute, JOLIE will directly retrieve it. However, in most cases, relative paths are more useful. In such cases JOLIE will try to resolve the given filepath following some rules, interrupting the search as soon as the first valid result is obtained:

- the relative path is resolved w.r.t. the same directory that contains the file with the include statement of interest;
- if the including file is contained within a special archive such as a JAR (Java Archive), the path is resolved w.r.t. the position of the including file in the archive;
- for each library path or archive passed as a command line parameter, the path is resolved w.r.t. it;
- the path is resolved w.r.t. the directory containing the installed JOLIE standard `include` library;
- for each inclusion path passed as a command line parameter, the path is resolved w.r.t. it.

3.5 Session management

A session represents an executing instance of a service behaviour. Sessions are one of the most important concepts in service-oriented computing: they allow for a service to be always available to multiple invocations. Focus on session management is particularly stressed in the JOLIE language. In listing 3.1 the service offering the `sum` operation is made reiteratively available by means of a `while` block; that is not an encouraged programming practice: a cleaner way to obtain this mechanic is using sessions.

3.5.1 Execution modalities, session starting and initialization

JOLIE provides three modalities for executing sessions:

- `single`: only a single session is executed; this is the default modality;
- `sequential`: sessions are executed sequentially, so a new session may start only when there is no currently executing session;
- `concurrent`: all sessions are executed in parallel, and new sessions are started as soon as they are requested.

Programmers can specify the desired modality through the `execution` instruction, in the deployment definition:

DeploymentInstruction ::= ...
| `execution` { **ExecutionModality** }
ExecutionModality ::= `single` | `sequential` | `concurrent`

A session is initiated when the first input operation statement programmed within the behaviour is invoked. The statement is executed and then the workflow that follows it is run.

Given that, a service offering multiple session creations through the sequential or the concurrent modality may still need to perform some kind of initialization, and such initialization may need to receive some input from

an external service. In those cases the programmer would not want these input statements to start any session, but simply be executed once during the initialization phase. The `init` blocks address this issue. Code inside these blocks is run before that inside the `main` procedure and can not start sessions.

3.5.2 Session state and synchronization

Each session has its own *local variable state*, thus the programmer generally does not need to worry about race conditions between sessions on variable accesses. Nevertheless, a *global variable state* is provided in order to share data among different sessions. The global variable state can be used by prefixing a variable path with keyword `global`, like in the following:

```
global.myGlobalVariable = 3; // A global variable
myLocalVariable = 1 // A local variable
```

Using the global state introduces the problem of managing concurrent access to global variables. This can be handled through the `synchronized` construct:

Process ::= ...
| `synchronized (` *id* `) {` **Process** `}`

which ensures that only one executing process at a time will enter any `synchronized` block sharing the same *id*.

The differentiation between local state and global state w.r.t. sessions is a peculiar approach to variable state handling. Indeed, in other more known languages variable state is usually dictated by the position of their declaration w.r.t. code blocks or function definitions. JOLIE, instead, does not require variables to be declared in advance and variables in the same sessions are always shared among all the activities of the same session. This approach stresses out the importance that JOLIE gives to a programming style that focuses on sessions and operation invocations, different from those based upon code block nesting or functions.

3.5.3 Correlation sets

JOLIE supports session correlation following the semantics of SOCK. Correlation sets are a mechanism for relating each incoming message to the session it is intended for. The correlation set of a service is to be specified in the deployment definition:

DeploymentInstruction	::=	...
	|	`cset` { **CorrSet** }
CorrSet	::=	**CorrVar** | **CorrVar , CorrSet**
CorrVar	::=	**VariablePath**
	|	**VariablePath : VariablePathList**
VariablePathList	::=	**VariablePath**
	|	**VariablePath VariablePathList**

In addition to the syntax provided in SOCK, JOLIE allows for the specification of multiple variable paths in which a correlation variable may occur. This is useful because the same correlation variable may be included in different parts of messages, depending on the invoked operation. In order to clarify this point, let us consider an example in which a service supports the management of simple chat rooms. Each chat room is identified by a univocal name. Invokers may start new chat rooms, close existing chat rooms and send messages to all the people in a chat room. The behavioural pseudocode and the correlation set definition for such a service may look like the following:

```
cset {
    adminToken,
    roomName:
        creationRequest.room.name
        message.roomName
        enterRequest.roomName
}
```

```
execution { concurrent }

main
{
    createRoom( creationRequest )( adminToken ) {
        getSecureToken@SecurityHelper()( adminToken )
    };
    keepRun = 1;
    while( keepRun ) {
        [ enterRoom( enterRequest )( clientToken ) {
            ... insert client in chat room ...
        } ] { nullProcess }
        [ sendMessage( message ) ] {
            ... send message to
                all clients in chat room ...
        }
        [ closeRoom( adminToken ) ] {
            keepRun = 0
        }
    }
}
```

Sessions are created by calling the `createRoom` operation, which generates a secure `adminToken` token and returns it to the invoker. Then, the program enters into a loop that ends only when the `closeRoom` operation is called by someone that knows the administration token. Sessions can also be referred to by their room name. A room name can appear in different variable paths, as specified by the `cset` definition, but they are all considered as the session room name.

Chapter 4

Advanced features

In this chapter more advanced mechanisms offered by JOLIE are shown, with particular emphasis on aspects regarding bindings and service architectures.

4.1 Dynamic port configuration

Input and output ports offer some configuration capabilities at runtime, which can be accessed by the behavioural code of a program. In particular, input ports allow for their protocol configuration to be read and/or written through some special variables. Output ports allow even more, permitting their location, protocol and protocol configuration to be changed.

Input port protocol configurations may be accessed through the `global.inputPorts.`*`inputPortName`*`.protocol` structure, where *inputPortName* is intended to be substituted with the name of the input port of interest. This feature is useful for protocols which may need to know some additional information at runtime for operating correctly, such as the HTTP protocol in some cases regarding the sending of responses.

The dynamicity offered by output ports is more expressive and is meant to be used so to implement a technique known as *rebinding*, explained in details in the following.

4.1.1 Rebinding and binding registries

The location and protocol (along with its configuration) of an output port represent its *binding* information. Binding information describes how to reach another service, in order to communicate with it. JOLIE comes with a type definition for bindings in its standard library:

```
type Binding:void {
    .location:string
    .protocol:string { ? }
}
```

Binding information of an output port can be accessed by means of a variable path starting with its name. For instance, the following would print the location and protocol name of output port `Printer`:

```
include "console.iol"

interface PrinterInterface {
OneWay:
    printText(string)
}

outputPort Printer {
Location: "socket://printerservice:8000/"
Protocol: sodep
Interfaces: PrinterInterface
}

main
{
    println@Console( Printer.location )();
    println@Console( Printer.protocol )()
}
```

where `println` is an operation for printing on the running console, offered by the `Console` service of the JOLIE standard library. Binding information may be entered at runtime by making simple assignments:

```
outputPort Printer {
Interfaces: PrinterInterface
}

main
{
    Printer.location =
        "socket://printerservice:8000/";
    Printer.protocol = "sodep"
}
```

The fact that JOLIE represents port information using such data structures paves the way for very elegant transmission of bindings. Let us consider an example where a binding registry offers a `getBinding` operation that is meant to return the binding information for contacting a service, where services are identified by name. Its interface is stored in a file, `Registry.iol`, to be included by client applications:

```
// Registry.iol

interface RegistryInterface {
RequestResponse:
    getBinding(string)(Binding)
}

outputPort Registry {
Location: "socket://registry.com:80/"
Protocol: soap
Interfaces: RegistryInterface
```

```
}
```

The code of a service implementing such an interface could look like the following:

```
// Registry.ol

include "Registry.iol"

execution { concurrent }

inputPort RegistryInput {
Location: "socket://registry.com:80/"
Protocol: soap
Interfaces: RegistryInterface
}

init
{
    with( global.bindings.LaserPrinter ) {
      .location = "socket://printerservice:8000/";
      .protocol = "sodep"
    };
    with( global.bindings.InkJetPrinter ) {
      .location = "socket://otherprinter:80/";
      .protocol = "soap"
    }
}

main
{
    getBinding( name )( global.bindings.(name) ) {
        nullProcess
```

```
    }
}
```

Finally, a client using such registry would call `getBinding` and receive its result on an output port variable path:

```
include "console.iol"
include "Registry.iol"

interface PrinterInterface {
OneWay:
    printText(string)
}

outputPort Printer {
Interfaces: PrinterInterface
}

main
{
    getBinding@Registry( "LaserPrinter" )( Printer );
    printText@Printer( "My text" )
}
```

4.1.2 Dynamic parallel composition

JOLIE offers a primitive for performing dynamic parallel compositions, i.e. parallel compositions whose width is determined at runtime. This is obtained by replicating a **Process** for a number of times that is given by an expression. The instruction that enables this behaviour is the `spawn` primitive:

Process ::= ...
| `spawn` `(` **VariablePath** `over` **Expression** `)`
SpawnInClause `{` **Process** `}`
SpawnInClause ::= `in` **VariablePath**
| ϵ

The `spawn` primitive creates a parallel composition by replicating the given **Process** by a number of times equal to the integer evaluation of the passed **Expression**. Each spawned process possesses its own local state, so that they do not interfere with each other. Because of this, an optional `in` clause is provided; the `in` clause causes the spawned processes to put the value of their local variable, pointed by the **VariablePath** immediately after the `in` keyword, in an element of the vector variable, pointed by the same path, of the activity that encloses the `spawn` block. Moreover, each spawned process is identified by an integer which goes from 0 to $n-1$, where n is the integer evaluation of the passed **Expression**. The spawned processes can read their own identifier by accessing the variable pointed by the first **VariablePath**.

Dynamic parallel composition is particularly useful when one needs to invoke multiple services whose number is not statically known. Consider, e.g., an activity that needs to perform a query to multiple travel agencies in order to get the best available offer. This can be easily implemented by means of a `spawn` block:

```
spawn( i over #bindings ) in travelPlan {
    TravelAgency << bindings[i];
    getTravelPlan@TravelAgency( request )
        ( travelPlan )
}
```

After executing this code one would obtain a `travelPlan` vector, which would have been populated by the spawned processes with their own single `travelPlan` results. The single result of a spawned processes identified by `i` can be accessed by referring to the element at position `i` in `travelPlan`.

4.2 Embedding

Embedding is a powerful mechanism for executing multiple services in the same virtual machine. A distinction is made between the *embedder* and the *embedded* service. The former is a service that embeds the latter. Embedding is very useful for handling the granularity of an SOA, for two reasons:

- services in the same virtual machine may communicate using fast local memory communication channels;
- embedding introduces a hierarchy of services, where the embedder is the parent service of the embedded ones; whenever a service terminates all its embedded services are recursively terminated.

The advantage of the first point lies in that one can build lightweight and reusable services that are designed to be embedded by more complex orchestrators, without influencing negatively performance as would be the case in more widespread technologies such as BPEL. The second point allows the programmer to design an orchestrator so to load all its dependencies explicitly at start-up, and leave the burden of terminating them when the orchestrator ends its execution to the JOLIE engine.

Embedding blurs the boundaries between the concepts of service and SOA: a single service may indeed embed an entire service-oriented architecture. Embedded services can, nonetheless, transparently continue to expose their interfaces to the outer world through their own input ports.

The syntax for embedding is:

DeploymentInstruction	::=	...
	|	`embedded` { **EmbeddingBlock*** }
EmbeddingBlock	::=	**EngineType** : **EmbeddedInstructionList**
EngineType	::=	`Jolie` | `Java`
EmbeddedInstructionList	::=	**EmbeddedInstruction**
	|	**EmbeddedInstruction** , **EmbeddedInstructionList**
EmbeddedInstruction	::=	`"` *string* `"` | `"` *string* `"` `in` *id*

EngineType expresses the engine needed to load the service to embed. JOLIE currently supports embedding services written in JOLIE itself or in Java [1]. In the case of a JOLIE service, one must point the filepath where its source code can be found. Command line parameters can also be passed before the filepath. In the case of a Java service, the fully qualified name of the class from which the service should be instantiated must be written.

Local memory communications are enabled by means of the `local` communication medium and the optional `in` clause of **EmbeddedInstruction**. In such cases no protocol definition is needed. In order to illustrate this point, let us consider the following example, where a simple `Echo` service gets embedded:

```
// Echo.ol

execution { concurrent }

interface EchoInterface {
RequestResponse:
    echo(string)(string)
}
```

[1] Experimental support for JavaScript is in development, but still unstable and as such not reported here.

```
inputPort EchoInput {
Location: "local"
Interfaces: EchoInterface
}

main
{
    echo( message )( message ) {
        nullProcess
    }
}
```

```
// Embedder.ol

include "console.iol"

interface EchoInterface {
RequestResponse:
    echo(string)(string)
}

outputPort Echo {
Interfaces: EchoInterface
}

embedded {
Jolie:
    "Echo.ol" in Echo
}

main
```

```
{
    echo@Echo( "Hello, world!" )( response );

    // Will print "Hello, world!"
    println@Console( response )()
}
```

4.2.1 Java services

JOLIE supports embedding Java code that follows its specifications for *Java Services*. Embedding Java Services is particularly useful for reusing existing Java code, perform some task where computational performance is important or interoperating with some existing legacy software. A Java Service can be written by extending the `JavaService` class, provided by the JOLIE runtime environment library (in package `jolie.runtime`). Most services of the JOLIE standard library are implemented in Java.

Each method of an embedded Java service is seen as an operation from the embedder. Services written in Java are automatically considered as executing with a concurrent modality. Commodity transformations are provided for basic data types such as `int`, `double` and `string`. Structured data must be handled, instead, with the `Value` class provided by the JOLIE runtime package. The `Value` class offers methods for accessing subnodes in a manner consistent with the language semantics of JOLIE.

The following is an example of a simple service which calculates and returns the length of a string:

```
package example;

import jolie.runtime.JavaService;

public class MyService extends JavaService
{
```

```
    public Integer length( String request )
    {
        return request.length();
    }
}
```

Such a service can be embedded and called as one would do with JOLIE services:

```
interface MyServiceInterface {
RequestResponse:
    length(string)(int)
}

outputPort MyService {
Interfaces: MyServiceInterface
}

embedded {
Java:
    "example.MyService" in MyService
}

main
{
    length@MyService( "Hi" )( l );
    println@Console( l )() // Will print 2
}
```

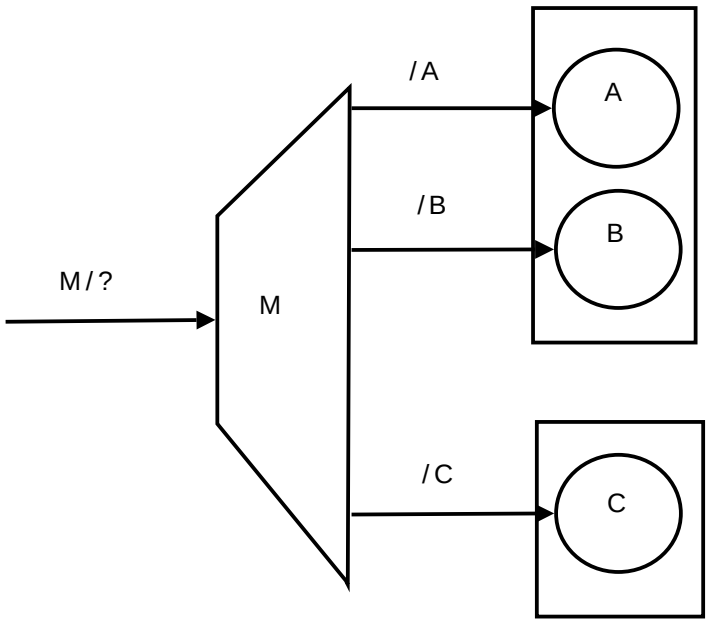

Figure 4.1: Service M redirects messages to services A, B and C depending on the target destination of the message (M/A, M/B or M/C).

4.3 Redirection

Redirection allows for the creation of a *master service* acting as a single communication endpoint to multiple services, called resources. The master service receives all the messages meant for the system that it handles. This is obtained by binding an input port of the master service to multiple output ports, each one identifying a service by means of a *resource name*. Invokers send messages to the master service specifying also the intended resource name. The main advantages of such an approach are:

- the possibility to provide a unique access point to the system clients. In this way the services of the system could be relocated and/or replaced transparently w.r.t. the clients;
- the possibility to provide transparent communication protocol transformations between the invoker and the master and the master and the rest of the system.

In order to understand the second advantage better, consider Fig. 4.1 and suppose that A speaks a certain protocol p_a. Now suppose that a client needs to interact with A, but it does know only a different protocol: p_m. The client could then call M with destination M/A using protocol p_m (known by M), and leave to M the task of transforming the call message into an instance of p_a before sending it to A.

The syntax for setting up a master service follows:

DeploymentInstruction ::= ...
| `inputPort` *id*
{ **PortInstruction***
InputPortInstruction* }

InputPortInstruction ::= `Redirects:`
RedirectionList

RedirectionList ::= **Redirection**
| **Redirection , RedirectionList**

Redirection ::= *id* `=>` *outputPortId*

where *outputPortId* ranges over output port identifiers and the form *id* `=>` *outputPortId* associates a resource name to an output port. The example in Fig. 4.1 can be implemented by means of that syntax:

```
outputPort ServiceA {
Location: "socket://www.somelocationA.com/"
Protocol: soap
Interfaces: InterfaceA
}

outputPort ServiceB {
Location: "socket://www.somelocationB.com/"
Protocol: sodep
Interfaces: InterfaceB
}
```

```
outputPort ServiceC {
Location: "socket://www.somelocationC.com/"
Protocol: http
Interfaces: InterfaceC
}

inputPort MasterInput {
Location: "socket://masterservice.com:8000/"
Protocol: sodep
Redirects: A => ServiceA, B => ServiceB, C => ServiceC
}
```

Calling a master service for one of its resources is done by introducing the resource name in the location used by the invoker, followed by the resource name separator `!/`:

```
outputPort A {
Location: "socket://masterservice.com:8000/!/A"
Protocol: sodep
Interfaces: InterfaceA
}
```

4.4 Aggregation

Aggregation is a composition of services where their interfaces are joined together and published as unique. Therefore, aggregation deals with the grouping of more services under the same interface. The mechanism is similar to redirecting, but there are not resource names visible from the point of view of the client; the client, instead, sees a unique service, the master one, which exhibits an interface by providing the functionalities of the resource services. Differently from redirecting, which maintains the different interfaces of each composed service separated, in this case the client looses the details

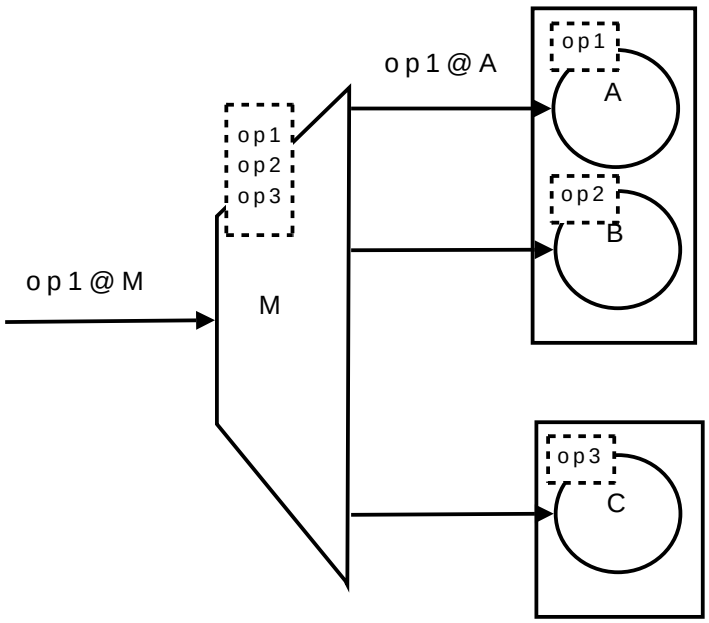

Figure 4.2: In aggregation the master service publishes the union of all the service interfaces it aggregates. Interfaces are here represented with dotted rectangles. The message on operation *op*1 to service M is actually redirected to service A.

of each single service used behind aggregation. The main advantage of such a composition approach deals with the possibility to completely hide the system components to the client.

The syntax for aggregation is:

InputPortInstruction ::= ...
| `Aggregates:` **OutputPortList**

OutputPortList ::= *outputPortId*
| *outputPortId* , **OutputPortList**

Thus, one could easily implement a scenario such as that represented in Fig. 4.2 with the following code:

```
outputPort ServiceA {
Location: "socket://www.somelocationA.com/"
Protocol: soap
Interfaces: InterfaceA
}

outputPort ServiceB {
Location: "socket://www.somelocationB.com/"
Protocol: sodep
Interfaces: InterfaceB
}

outputPort ServiceC {
Location: "socket://www.somelocationC.com/"
Protocol: http
Interfaces: InterfaceC
}

inputPort MasterInput {
Location: "socket://masterservice.com:8000/"
Protocol: sodep
```

```
Aggregates: ServiceA, ServiceB, ServiceC
}
```

An input port that makes use of aggregation can still expose an interface of its own. If conflicts are present, i.e. the service interface contains an operation that is exposed by one of the aggregated output ports, they are resolved in favour of the direct input port interface.

4.5 Dynamic system composition

The aforementioned service composition techniques (dynamic binding, embedding, redirection and aggregation) can be used statically or at runtime. In the static case all the services are composed before their execution and the composition never changes during the execution of all the system. On the contrary, if the composition of the system changes at runtime, the system is said to be *dynamically composed*. Dynamic composition is strictly related to the concept of *service mobility*. Service mobility deals with the representation of a service in some data format, its transmission from one service to another and then its execution in the service container of the receiver. Note that this does not imply that a service can be moved while it is executing: JOLIE provides mechanisms for moving service definitions, but session and state mobility must still be implement manually by the programmer. In the following some cases of dynamic composition are, for the sake of brevity, briefly described. They can all be implemented using operations provided by the JOLIE standard library through the `Runtime` service, which is distributed with the language runtime environment and is publicly consultable.

Dynamic embedding. Let us consider a service which needs to receive software updates for a certain functionality. One may encapsulate that functionality in an embedded service. Then, when a software update is issued, the embedder service may unload the embedded one, receive the updated

service to embed and dynamically embed the received service.

Dynamic redirecting and aggregation. Let us consider the case that a resource service faults or needs maintenance without affecting the service availability from the client point of view. It is sufficient to install a fresh resource service and to put it in the master service in place of the faulty one.

Chapter 5

Fault and compensation handling

This chapter presents the error handling mechanisms provided by the JOLIE language. Error handling in service-oriented applications can be quite complicated, due to their concurrent nature. For this reason, a foundational study on SOCK has been conducted. The key concepts behind error handling in SOCK and JOLIE are reported first, followed by a description of the theoretical study performed w.r.t. SOCK. Finally, the constructs offered by JOLIE for fault handling are presented.

5.1 Key concepts

Fault handling in SOC involves four basic concepts: *scope*, *fault*, *termination* and *compensation*. A scope is a process container denoted by a unique name and able to manage faults. A fault is a signal raised by a process towards the enclosing scope when an error state is reached, in order to allow for its recovery. Termination and compensation are mechanisms exploited to recover from errors. Termination is triggered when a scope must be smoothly stopped, whereas compensation is triggered to undo the effect of a scope whose execution has already successfully terminated. Recovery mechanisms

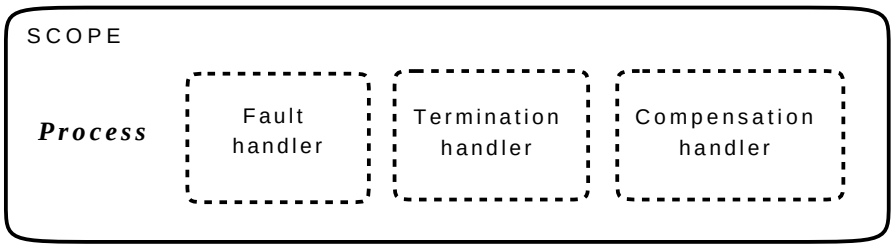

Figure 5.1: Handlers in a scope

are implemented by exploiting *handlers* which contain processes to be executed when faults, terminations or compensations are triggered. Handlers are defined within a scope which represents the execution boundaries for their application. There are three kinds of handlers: *fault handlers*, *termination handlers* and *compensation handlers*. Fault handlers are executed when a fault is triggered by the internal process of the scope, termination handlers are executed when a scope is reached by a fault raised by an external process and, finally, compensation handlers can be explicitly invoked by another handler for recovering the activities of a child scope whose computation has already successfully finished. Fig. 5.1 shows all the elements composing a scope. A language managing error recovery via statically defined scopes (such as BPEL) should provide a primitive like $\texttt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$ where q is the scope name, P the executing process and $\mathcal{FH}$, $\mathcal{TH}$ and $\mathcal{CH}$ are, respectively, the fault, termination and compensation handlers. When a fault is raised, it is propagated and it causes the termination of all the other activities inside the same scope. After that, if the fault handler for that fault is defined, the scope executes it, otherwise it forwards the fault to the outer scope. It is worth noting that a terminating activity could be a scope, and in this case its termination handler should be executed. Also, some linguistic primitive, such as `comp(q)`, can be used to require the execution of the compensation handler of the scope named q. Fig. 5.2 provides an intuitive representation of handler mechanisms where numbers represent ordered events and *stm1,stm2,...,stmn* represent a list of generic statements. A scope A encloses a generic process P and two scopes B and C. At 1 scope C fin-

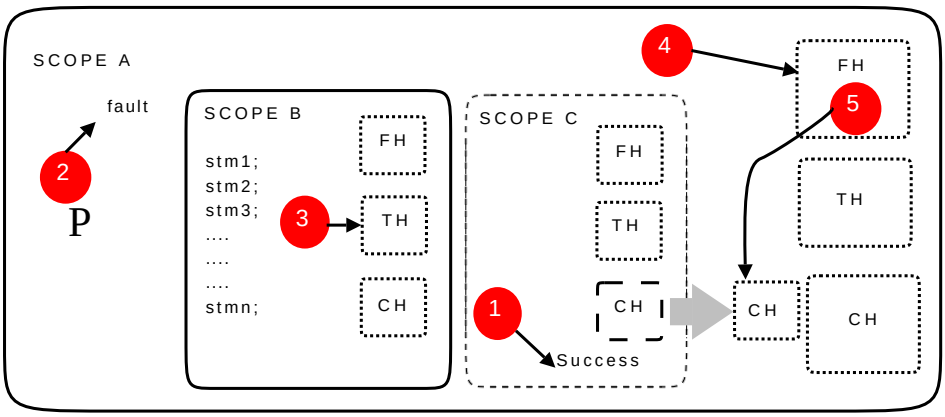

Figure 5.2: Handler mechanisms

ishes successfully by promoting its compensation handler to be executable by the enclosing scope A. At 2, process P raises a fault which is propagated to scope B. B is assumed to be still in execution when reached by the fault so, at 3, it executes its termination handler and terminates. At 4 the fault handler of scope A is executed and, at 5, it compensates scope C (supposing that the handler specifies so).

In some cases static declaration of handlers is not enough to easily model a given scenario. Let us consider the following pseudo-code:

$$\texttt{scope}_q(\texttt{while}(i < 100)(\texttt{if } i\%2 = 0 \texttt{ then } P \texttt{ else } Q), \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$$

Scope q contains a loop which executes 100 cycles. Even cycles execute process P whereas odd cycles execute process Q. If scope q is reached by a fault, in order to correctly recover its activities, it has to remember their exact sequence and recover them in the desired order. One can use some bookkeeping variables, but as far as the complexity of the code increases the bookkeeping becomes more complex and error-prone. In order to address this problem JOLIE makes use of *dynamic handling*, which allows to update handlers as far as the computation progresses. Each scope contains a function $\mathcal{H}$ associating fault handlers to fault names and termination and compensation handlers to scope names.

Technically, dynamic handling is addressed by an *installing* primitive,

inst($\mathcal{H}$), which updates the current handler function with $\mathcal{H}$. Thus, the handler code can be updated depending on the current state of the scope. The example above could be rewritten by exploiting the dynamic handler mechanism as:

$\texttt{scope}_q(\texttt{while}(i < 100)$
$\quad \texttt{if } i\%2 = 0 \texttt{ then } P; \text{inst}([q \mapsto P'; cH])$
$\quad \texttt{else } Q; \text{inst}([q \mapsto Q'; cH])$
$, \mathcal{H})$

where cH allows to recover the previously installed handler with the same name. In this case when P is executed the termination handler is updated with process P', which specifically recovers process P (inst($[q \mapsto P'; cH]$)), whereas if Q is executed the termination handler is updated with Q'. When reached by a fault, scope q executes the last installed termination handler, thus recovering the whole sequence of activities. Different strategies can easily be programmed. Notice that in the example above it should never be the case that an execution of P has been completed and its compensation has not been installed, since otherwise the compensation would not be up-to-date. This can be obtained in the dynamic scenario by giving precedence to the inst primitive, while the same can not be done for the bookkeeping code needed in the static framework.

In this scenario, when a scope successfully terminates, the last defined termination handler becomes its compensation handler. It is worth noting that there is no ambiguity between the two handlers since they are triggered in different ways. Termination handler is executed by the scope itself which stops its normal code (in Fig. 5.2 scope B stops and executes its termination handler), whereas the compensation handler is always executed by the enclosing scope (in Fig. 5.2 the fault handler of the scope A executes the compensation of scope C) . This allows also to trivially simulate the static approach with the dynamic one: the construct $\texttt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$ can be simply rephrased as $\texttt{scope}_q(\text{inst}(\mathcal{FH}); \text{inst}(\mathcal{TH}); P; \text{inst}(\mathcal{CH}))$ in which the fault and termination handlers are installed before the execution of the

activity, and the compensation handler at the end.

5.2 Foundations for dynamic error handling

In this section the SOCK calculus is extended by adding to its service behaviour layer the aforementioned primitives for fault and compensation handling.

Syntax. The following additional sets are used: $Faults$, ranged over by f, for faults, and $Scopes$, ranged over by q, for scope names. $q_\perp$ ranges over $Scopes \cup \{\perp\}$, whereas u ranges over $Faults \cup Scopes \cup \{\perp\}$. Here $\perp$ is used to specify that an handler is undefined. $\mathcal{H}$ denotes a function from $Faults$ and $Scopes$ to processes extended with $\perp$, i.e. $\mathcal{H} : Faults \cup Scopes \to SC \cup \{\perp\}$. In particular, the function associating P_i to u_i for $i \in \{1, \ldots, n\}$ is written as $[u_1 \mapsto P_1, \ldots, u_n \mapsto P_n]$. The extended syntax for processes is defined in Table 5.1. Note that the syntax for outputs $\overline{\epsilon}$ has changed, since now the solicit-response $\overline{o_r}@z(\vec{y}, \vec{x}, \mathcal{H})$ includes a handler update $\mathcal{H}$, whose purpose will be clarified later.

In addition to the new solicit-response primitive there are five new static constructs, and other auxiliary constructs to help the definition of the semantics. Handlers are installed by inst($\mathcal{H}$), where $\mathcal{H}$ is a partial function from fault and scope names to processes: $\mathcal{H}(f) = \perp$ is used to specify that $\mathcal{H}$ is undefined on fault name f. $\{P\}_q$ defines a scope named q and executing process P. This is a shortcut for $\{P : \mathcal{H}_0 : \perp\}_q$ where $\mathcal{H}_0$ is the function that evaluates to $\perp$ for all fault names (i.e., at the beginning no fault handler is defined) and to $\mathbf{0}$ for all scope names (i.e., the default termination or compensation handler has no effect). The third parameter is the name of a handler waiting to be executed: at the beginning no handler has to be executed, thus $\perp$ is used. Primitives throw(f) and comp(q) respectively raises fault f and asks to compensate scope q. cH is a placeholder for the previously installed handler with the same name, to be used inside a handler update.

Well-formedness rules. Informally, comp(q) and cH occur only within

$$\epsilon ::= o(\vec{x}) \mid o_r(\vec{x}, \vec{y}, P) \qquad \overline{\epsilon} ::= \overline{o}@z(\vec{y}) \mid \overline{o_r}@z(\vec{y}, \vec{x}, \mathcal{H})$$

$P, Q, \ldots ::=$	ϵ	*input*
	$\overline{\epsilon}$	*output*
	$\ldots$	*other standard ops.*
	$\{P\}_q$	*scope* (for $\{P : \mathcal{H}_0 : \bot\}_q$)
	$\text{inst}(\mathcal{H})$	*install handler*
	$\text{throw}(f)$	*throw*
	$\text{comp}(q)$	*compensate*
	cH	*current handler*
	$Exec(P, o_r, \vec{y}, l)$	*Request-Response execution*
	$\{P : \mathcal{H} : u\}_{q_\bot}$	*active scope*
	$o_r(\vec{x}, \mathcal{H})$	*response in solicit*
	$o_r\langle \vec{x}, \mathcal{H} \rangle$	*dead response in solicit*
	$\langle P \rangle$	*protection*
	$\overline{o_r}!f@l$	*fault output*

Table 5.1: Service behaviour syntax with faults

handlers, and q can only be a child of the enclosing scope. Also, for each $\text{inst}(\mathcal{H})$, $\mathcal{H}$ is undefined on all scope names q but the one of the nearest enclosing scope, i.e. a process can define the termination/compensation handler only for its own scope. Finally, it is assumed that scope names are unique.

Semantics. As before, in order to define the semantics an extended syntax is exploited. There $\{P : \mathcal{H} : u\}_{q_\bot}$ is an active scope. An active scope may have a handler function $\mathcal{H}$ specifying the installed handlers. Also, u is the name of a handler waiting to be executed, or $\bot$ if no handler is waiting to be executed. This is needed, for instance, when scope q is killed while waiting for the answer of a Request-Response interaction: the termination handler for q has to be executed, thus q is written as third parameter. However, the

termination handler is not executed immediately, but the answer from the Request-Response is waited for. This answer may update the termination handler. When the termination handler has to be executed, the updated version is used. When a scope has failed its execution, either because it has been killed from a parent scope, or because it has not been able to catch and manage an internal fault, it reaches a zombie state. This is identified since the name of the scope becomes $\perp$. Scopes in a zombie state are no more able to throw faults. This ensures that each scope may throw at most one fault. Also, $o_r(\vec{x}, \mathcal{H})$ is used to wait for the response in a solicit-response interaction. $\mathcal{H}$ is installed iff a non faulty response is received, allowing to program the compensation for the remote activity. If the remote activity has failed, no compensation for it is required. $o_r\langle\vec{x}, \mathcal{H}\rangle$ is the corresponding zombie version, which cannot throw faults. This is created when the normal version is killed because of an external fault, again to ensure that no fault is raised by dead activities. $\langle P\rangle$ executes P in a protected way, i.e. not influenced by external faults. This is needed to ensure that recovery from a fault is completed even if another fault happens. Also, $Exec(P, o_r, \vec{y}, l)$ is a running Request-Response interaction (as for the calculus without faults). Finally, $\overline{o_r}!f@l$ notifies fault f to the client (located at l) of a Request-Response pattern. This is created when an executing Request-Response is killed because of a local fault.

In the semantics, in addition to the structured actions introduced in Chapter 2, the following unstructured actions are used:

$$\{th(f), cm(q, P), inst(\mathcal{H})\}$$

They represent, respectively, the propagation of fault f, the check that the compensation code for scope q is P, and the request to apply handler update $\mathcal{H}$.

The service behaviour calculus does not deal with the actual values of variables and locations but it models all the possible execution paths for all the possible variable values and locations. The semantics follows this idea by means of an infinite set of actions where external inputs, external outputs

and assignment actions report all the value substitutions for both variables and locations except the actions $\overline{o}@l(\vec{v}/\vec{x})$ and $\overline{o_r}@l(\vec{v}/\vec{x},\vec{y},)$ where locations are defined. Formally, let *Act* be the set of actions, ranged over by γ, defined as follows:

$$
\begin{aligned}
&Act = In \cup Out \cup Internal\\
&In = \{o(\vec{v}/\vec{x}), o_r(\vec{v}/\vec{x},\vec{y},P)@l\}\\
&Out = \{\overline{o}@l/z(\vec{v}/\vec{x}), \overline{o}@l(\vec{v}/\vec{x}), \overline{o_r}@l/z(\vec{v}/\vec{x},\vec{y},,)\overline{o_r}@l(\vec{v}/\vec{x},\vec{y},\})\\
&Internal = \{s, \bar{s}, \tau, th(f), cm(q,P), inst(\mathcal{H})\}
\end{aligned}
$$

Definition 5.2.1 (Service behaviour layer semantics)**.** $\rightarrow \subseteq SC \times Act \times SC$ *is the least relation which satisfies the rules of Tables 2.2 (where* $th(f)$ *is supposed to never occur as label) and 5.2, and closed w.r.t. structural congruence* $\equiv$*, the least congruence relation satisfying the axioms in Table 2.3.*

The rules in Table 5.2 define the semantics of scopes, faults and compensations. Operator $\boxplus$ is used for updating the handler function:

$$
(\mathcal{H} \boxplus \mathcal{H}')(u) = \begin{cases} (\mathcal{H}'(u))[\mathcal{H}(u)/cH] \text{ if } u \in \mathrm{Dom}(\mathcal{H}') \cap \mathrm{Dom}(\mathcal{H}) \\ (\mathcal{H}'(u))[\mathbf{0}/cH] \text{ if } u \in \mathrm{Dom}(\mathcal{H}'), u \notin \mathrm{Dom}(\mathcal{H}) \\ \mathcal{H}(u) \text{ otherwise} \end{cases}
$$

where inst is a binder for cH, i.e. substitutions are not applied inside the inst primitive.

Intuitively, the above definition means that handlers in $\mathcal{H}'$ replace the corresponding handlers in $\mathcal{H}$, and occurrences of cH in the new handlers are replaced by the old handlers with the same name. For instance, $\mathrm{inst}([q \mapsto P|cH])$ adds P in parallel to the old handler for q. Furthermore, $\mathrm{cmp}(\mathcal{H})$ denotes the part of $\mathcal{H}$ dealing with terminations/compensations, i.e. $\mathrm{cmp}(\mathcal{H}) = \mathcal{H}|_{Scopes}$.

Rules Solicit and Solicit-Response replace the corresponding rules in Table 2.2, ensuring that when the answer is received, handler update $\mathcal{H}$ is installed. The handler update is not performed if a fault answer is received

(see rule RECEIVE-FAULT). The internal process P of a scope can execute thanks to rule SCOPE. Fault and compensation handlers are installed in the nearest enclosing scope by rules ASKINST and INSTALL. According to rule SCOPE-SUCCESS, when a scope successfully ends, its compensation handlers are propagated to the parent scope. This is needed to allow the parent scope to compensate the finished child. Compensation handlers of subscopes are propagated too, to allow to recursively compensate them. Compensation execution is required by rule COMPENSATE. The actual compensation code Q is guessed, and the guess is checked by rule COMPENSATION. When the compensation is executed, the corresponding handler is removed, so to ensure that it is not possible to compensate twice the same activity. Faults are raised by rule THROW. A fault is caught by rule CATCH-FAULT when a scope defining the corresponding handler is met. The name of the handler is stored in the third component of the scope construct: the handler is executed only after the activities in P' have been completed. These include, for instance, waiting for response messages in Request-Response interactions, terminating subscopes, and terminating internal error recovery. This is managed by the rules for fault propagation THROW-SYNC, THROW-SEQ, RETHROW and THROW-REEXEC, and by partial function *killable* (see Table 5.3). Function *killable* is applied to parallel components by rule THROW-SYNC. This has a double aim. On the one hand it guarantees that when a fault is thrown there is no pending handler update, i.e. it gives priority to handler update w.r.t. fault processing. This ensures that handlers are always up-to-date, and solves the race condition issues discussed in 5.1. Technically this is obtained by making $killable(P, f)$ undefined (and thus rule THROW-SYNC not applicable) if some handler installation is pending in P. On the other hand $killable(P, f)$ computes the activities that have to be completed before the handler is executed. In particular, when a sub-scope is terminated, its termination handler is marked as next handler to be executed (see the definition of function *killable* for scopes). The latter may substitute a previously marked fault handler, following the intuition that a request of termination has prior-

ity w.r.t. an internal activity such as fault processing. Also, if an *Exec* (i.e., an ongoing Request-Response computation) is terminated then the fault is notified to the partner (this is why function *killable* needs as parameter the name f of the fault). Finally, a receive waiting for the answer of a solicit-response is preserved, thus preserving the pattern of communication, but changed to its zombie version, ensuring that no other faults will be thrown. The $\langle P \rangle$ operator (defined by rule PROTECTION) guarantees that the enclosed activity will not be killed by external faults (because of the fifth rule in the definition of function *killable*). Rule SCOPE-HANDLE-FAULT executes a handler for a fault. This is done only after the activities discussed above have terminated. The fault handler is removed from the function $\mathcal{H}$ in order to allow throw primitives for the same fault in the handler to propagate the fault to the outer scope. Note that a scope that has handled an internal fault can still end with success. Instead, a scope that has been terminated from the outside is in zombie state. It can execute its termination handler thanks to rule SCOPE-HANDLE-TERM, and then terminate with failure (thus discarding its compensation handlers) using rules SCOPE-FAIL. Similarly, a scope enters the zombie state when reached by a fault it cannot handle, as specified by rule RETHROW. The fault is propagated up along the scope hierarchy. Zombie scopes cannot throw faults any more, since rule IGNORE-FAULT has to be applied instead of RETHROW. Rule IGNORE-FAULT is necessary only for faults thrown by handlers, since no other fault can be generated by a zombie scope. When an executing Request-Response is reached by a fault, it is transformed into a fault notification (see the definition of function *killable*). Fault notification is executed by rule SEND-FAULT, and it will interact with the waiting receive thanks to rule RECEIVE-FAULT. When received, the fault is ready to be rethrown at the client side, where it is treated as a local fault. If the receive is in zombie state instead, the fault is discarded (rules DEAD-SOLICIT-RESPONSE and DEAD-RECEIVE-FAULT are used instead of rules SOLICIT-RESPONSE and RECEIVE-FAULT).

5.3 Dynamic error handling in JOLIE

JOLIE offers the aforementioned fault handling mechanisms by means of a new set of instructions, each one resembling the primitives that have been introduced in SOCK:

Process	::=	...	
	|	`throw(` *f* `)`	*Throw*
	|	`throw(` *f* `,` **VariablePath** `)`	*Throw (w/ data)*
	|	`install(` **Handlers** `)`	*Install*
	|	`comp(` *s* `)`	*Compensate*
	|	`cH`	*Current handler*
	|	`scope(` *s* `)` { **Process** }	*Scope*
OutputStatement	::=	...	
	|	`op@OPort(` **VariablePath** `)`	
		`(` **VariablePath** `)`	
		`[` **Handlers** `]`	*Solicit-Response*
Handlers	::=	**Handler** | **Handler Handlers**	
Handler	::=	**FaultNameList => Process**	
FaultNameList	::=	*f* | *f* **FaultNameList**	

where *f* ranges over fault identifiers and *s* over scope identifiers. *Throw* raises a fault signal that can be equipped with extra data. *Install* performs a handler update in the enclosing scope. *Compensate* compensates a successfully finished scope. `cH` is a placeholder for referring to the previously installed handler. *Scope* is the construct for creating scopes. Finally, the syntax for performing solicit-response calls is extended with the possibility to install handlers, following the updated semantics of SOCK.

The introduction of faults influences the grammar for defining interfaces, too. In particular, Request-Response operations must now declare what faults they may throw to callers:

RequestResponseOp	::=	...
	|	*id* (**OpMessageType**)
		(**OpMessageType**)
		`throws` **FaultDeclarationList**
FaultDeclarationList	::=	**FaultDeclaration**
	|	**FaultDeclaration FaultDeclarationList**
FaultDeclaration	::=	*f* (**OpMessageType**)

An example showing how to define a service that may throw a fault follows:

```
type DivideRequest:void {
    .x:double
    .y:double
}

interface MyInterface {
RequestResponse:
    divide(DivideRequest)(double)
        throws DivideByZero(void)
}

inputPort MyInput {
Location: "socket://myinput.com:8000"
Protocol: sodep
Interfaces: MyInterface
}

main
{
    divide( request )( response ) {
        if ( request.y == 0 ) {
            throw( DivideByZero )
```

```
        };
        response = request.x / request.y
    }
}
```

JOLIE supports all the concepts and semantics for dynamic handling that have been exposed for SOCK. In the following the main features offered by the JOLIE implementation are analyzed.

Automatic fault transmission – As in SOCK, faults that are thrown from inside a Request-Response operation are automatically transmitted to the caller. This is particularly useful in practical application programming, because of the compositional nature of services; a service usually has to call another service in order to compute the response for a request. This generates a request chain that lasts until services that do not need to compose other services to answer their own requests are met. In such a scenario, automatic fault transmission is very useful because if an uncaught fault occurs in the chain the initial client receives the same fault that occurred (in case there are no renamings performed through rethrowing).

Dynamic code generation – The `cH` element implies that the language must be able to generate behavioural code dynamically. Consider the following example:

```
1) scope( s ) {
2)  install( f => i = i + 2 );
3)  install( f => i++; cH )
4) }
```

In (3) the install instruction contains a reference to the current handler. In order to execute the instruction correctly, JOLIE must first replace cH; so, the install instruction that gets executed at (3) is:

```
install( f => i++; i = i + 2 )
```

The code (in this case `i = i + 2`) is said to be *dynamically generated* by the

interpreter at the time of installation.

Actual programming experience showed that dynamic code generation must pay particular care to the evaluation of expressions: it is important, for the programmer, to be able to refer to the variable state at the time of handler installation. JOLIE offers this feature by means of the ^ operator, which can be used to prefix a variable and *freeze* its state in a handler that is going to be installed. In order to understand this concept, consider the following JOLIE code:

```
scope( s ) {
    for( i = 0, i < 3, i++ ) {
        install( f => println@Console( ^i )(); cH )
    };
    throw( f )
}
```

The program cycles over the `i` variable, and once it completes the `for` block it throws fault `f`, thus causing the installed fault handler to be executed. At each iteration, the `for` body updates the fault handler for `f` by prefixing a console output of `^i` to the currently installed handler; at each installation, the ^ operator replaces the value of `i` with its current value. The final handler for `f`, just before the throw instruction is reached, results then as:

```
println@Console( 2 )();
println@Console( 1 )();
println@Console( 0 )()
```

Structured fault data – JOLIE supports the association of structured data to a fault signal. This ability can be used to attach additional information to a fault, that can be retrieved and used later in the fault handler execution. In order to do this, JOLIE extends the throw instruction to support an optional parameter: `throw(` *f*, **VariablePath** `)`. This new primitive attaches the data pointed by **VariablePath** to fault signal *f* and then raises the latter. In the following a usage example is provided.

```
scope( s ) {
    install( f =>
        // This will print "Hello, world!"
        println@Console( s.f.message )()
    );
    data.message = "Hello, world!";
    throw( f, data )
}
```

Note that in order to refer to the fault data of `f` the scope name is exploited: `s.f.message`. This is due to the fact that in SOCK and JOLIE variables are shared, so if two scopes in parallel receive the same fault by their internal activities their respective fault data must be stored in two different variables to avoid a memory race condition. In order to do so, the name of the scope receiving the fault is used as a prefix. Note also that fault data is transparently transmitted over the network in case of faults thrown inside a Request-Response operation execution.

Basic safety properties – The dynamic fault handling mechanism defines some basic properties that are always assured, reported in [19]. JOLIE respects them and the programmer can make use of them in his or her reasoning about an orchestrator behaviour. The main properties are:

1. a scope ends successfully *if* it does not throw any fault upstream, i.e. its internal process does not throw any fault or the scope handles all of the faults thrown by its internal process;

2. a scope installs its compensations in the parent scope *iff* it ends successfully;

3. a scope that is terminated by a sibling parallel process (i) does not end successfully and (ii) does not throw any fault upstream anymore;

4. if a Solicit-Response process starts (i.e. it sends a request message) it always waits for the response;

5. if a Request-Response process starts (i.e. it receives a request message) it always supplies a response to the caller, be it a normal message or a fault.

Properties (1), (2) and (3) offer to the programmer the means to safely predict the behaviour of a scope. Properties (4) and (5) ensure that the Request-Response pattern is always respected, even when the program has to deal with fault handling.

Install statement priority – One of the most important aspects of dynamic handling is that the install primitive has priority w.r.t. fault processing. This introduces the necessary determinism to assure that fault handling behaviour is predictable by the programmer. JOLIE implements this mechanism exploiting its internal execution architecture, the Object-Oriented Interpretation Tree (OOIT), which will be detailed in Chapter 6. Consider the following code:

```
scope( s ) {
    throw( f )
    |
    install( f =>
        println@Console( "Hello, world!" )()
    )
}
```

where the behaviour is composed by two processes in parallel: the former throws a fault `f`, whereas the latter installs a fault handler for `f`. This workflow is internally represented by the OOIT in Fig. 5.3. Basically, every OOIT node is responsible for implementing a specific SOCK semantic rule. Fault signals are propagated upwards in the tree. When the fault signal `f` reaches the `|` node (i.e. the node representing the parallel composition), the latter informs every other child node that the parallel composition is now in a fault handling situation and waits for their confirmation. Normally, a node aborts its execution and returns immediately, but this is not the case for a

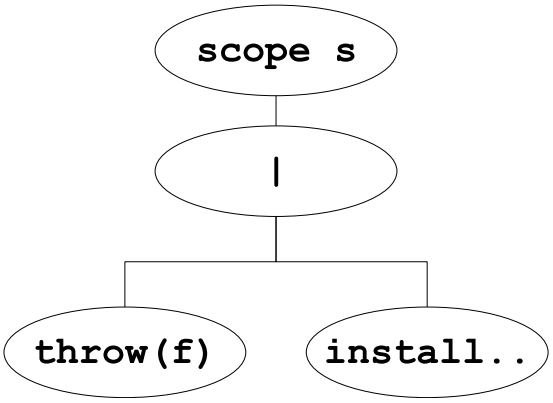

Figure 5.3: OOIT scope representation

node that has to perform a handler installation: an *install* node returns its confirmation to the | node only after actually performing the installation. Thus, the parallel composition is forced to wait for the handler installation and it propagates the fault signal to the `scope(s)` node only afterwards.

(Solicit)
$$\overline{o_r}@z(\vec{x},\vec{y},\mathcal{H}) \xrightarrow{\uparrow\overline{o_r}(\vec{v})@l(l/z,\vec{v}/\vec{x}:\emptyset)} o_r(\vec{y},\mathcal{H})$$

(Solicit-Response)
$$o_r(\vec{x},\mathcal{H}) \xrightarrow{\downarrow o_r(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathrm{inst}(\mathcal{H})$$

(Scope)
$$\frac{P \xrightarrow{a} P' \quad a \neq inst(\mathcal{H}), cm(q',\mathcal{H}')}{\{P:\mathcal{H}:u\}_{q_\perp} \xrightarrow{a} \{P':\mathcal{H}:u\}_{q_\perp}}$$

(AskInst)
$$\mathrm{inst}(\mathcal{H}) \xrightarrow{inst(\mathcal{H})} \mathbf{0}$$

(Throw)
$$\mathrm{throw}(f) \xrightarrow{th(f)} \mathbf{0}$$

(Compensate)
$$\mathrm{comp}(q) \xrightarrow{cm(q,Q)} Q$$

(Install)
$$\frac{P \xrightarrow{inst(\mathcal{H})} P'}{\{P:\mathcal{H}':u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P':\mathcal{H}'\boxplus\mathcal{H}:u\}_{q_\perp}}$$

(Scope-Success)
$$\{\mathbf{0}:\mathcal{H}:\perp\}_q \xrightarrow{inst(\mathrm{cmp}(\mathcal{H}))} \mathbf{0}$$

(Scope-Fail)
$$\{\mathbf{0}:\mathcal{H}:\perp\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \mathbf{0}$$

(Scope-Handle-Fault)
$$\{\mathbf{0}:\mathcal{H}:f\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(f):\mathcal{H}\boxplus[f\mapsto\perp]:\perp\}_{q_\perp}$$

(Compensation)
$$\frac{P \xrightarrow{cm(q,Q)} P', \mathcal{H}(q) = Q}{\{P:\mathcal{H}:u\}_{q'_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P':\mathcal{H}\boxplus[q\mapsto\mathbf{0}]:u\}_{q'_\perp}}$$

(Scope-Handle-Term)
$$\{\mathbf{0}:\mathcal{H}:q\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(q):\mathcal{H}\boxplus[q\mapsto\mathbf{0}]:\perp\}_\perp$$

(Protection)
$$\frac{P \xrightarrow{a} P'}{\langle P\rangle \xrightarrow{a} \langle P'\rangle}$$

(Throw-Sync)
$$\frac{P \xrightarrow{th(f)} P', killable(Q,f) = Q'}{P|Q \xrightarrow{th(f)} P'|Q'}$$

(Throw-Seq)
$$\frac{P \xrightarrow{th(f)} P'}{P;Q \xrightarrow{th(f)} P'}$$

(Catch-fault)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) \neq \perp}{\{P:\mathcal{H}:u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P':\mathcal{H}:f\}_{q_\perp}}$$

(Ignore-fault)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P:\mathcal{H}:u\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \{P':\mathcal{H}:u\}_\perp}$$

(ReThrow)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P:\mathcal{H}:u\}_q \xrightarrow{th(f)} \langle\{P':\mathcal{H}:\perp\}_\perp\rangle}$$

(Throw-RExec)
$$\frac{P \xrightarrow{th(f)} P'}{Exec(P,o_r,\vec{y},l) \xrightarrow{th(f)} P'|\,\langle\overline{o_r}!f@l\rangle}$$

(Send-Fault)
$$\overline{o_r}!f@l \xrightarrow{\overline{o_r}(f)@l(\emptyset:\emptyset)} \mathbf{0}$$

(Receive-Fault)
$$o_r(\vec{x},\mathcal{H}) \xrightarrow{o_r(f)(\emptyset:\emptyset)} \mathrm{throw}(f)$$

(Dead-Solicit-Response)
$$o_r\langle\vec{x},\mathcal{H}\rangle \xrightarrow{\downarrow o_r(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathrm{inst}(\mathcal{H})$$

(Dead-Receive-Fault)
$$o_r\langle\vec{x},\mathcal{H}\rangle \xrightarrow{o_r(f)(\emptyset:\emptyset)} \mathbf{0}$$

Table 5.2: Rules for service behaviour layer: fault and compensation rules

$$
\begin{aligned}
&killable(\{P : \mathcal{H} : u\}_q, f) = \langle \{killable(P, f) : \mathcal{H} : q\}_\perp \rangle \text{ if } P \not\equiv \mathbf{0} \\
&killable(P \mid Q, f) = killable(P, f) \mid killable(Q, f) \\
&killable(P; Q, f) = killable(P, f) \text{ if } P \not\equiv \mathbf{0} \\
&killable(Exec(P, o_r, \vec{y}, l), f) = killable(P, f) \mid \langle \overline{o_r}! f@l \rangle \\
&killable(\langle P \rangle, f) = \langle P \rangle \text{ if } killable(P, f) \\
&killable(o_r(\vec{y}, \mathcal{H}), f) = \langle o_r \langle \vec{y}, \mathcal{H} \rangle \rangle \\
&killable(P, f) = \mathbf{0} \text{ if } P \in \{\mathbf{0}, \epsilon, \overline{\epsilon}, x := e, if\ \chi\ then\ P\ else\ Q, \\
&\qquad\qquad while\ \chi\ do\ (P), \overline{o_r}! f'@l, o_r \langle \vec{y}, \mathcal{H} \rangle, \\
&\qquad\qquad \textstyle\sum_{i \in W} \epsilon_i; P_i, \mathrm{throw}(f), \mathrm{comp}(q)\}
\end{aligned}
$$

Table 5.3: *killable* function

Chapter 6

Implementation

This chapter describes the architecture of the reference implementation of the JOLIE language. Particular care has been put into making the final result easily extendable.

JOLIE is implemented through an interpreter written in the Java language. Source code gets parsed and transformed into objects implementing the desired semantics. These objects are organized into a tree, called OOIT (Object-Oriented Interpretation Tree), which is run inside a Runtime Environment supporting its execution. A separate component, called Communication Core, is used in order to perform communications. The JOLIE interpreter resulting architecture is thus composed by four main components, here summarily described and then more deeply analyzed in the following sections.

- *Runtime Environment*: it is responsible for instantiating the other components and supporting the execution of the OOIT.

- *Parser*: it reads the input program and generates the OOIT.

- *Object-Oriented Interpretation Tree (OOIT):* a tree of objects that implements the execution of the semantic rules relative to the input program.

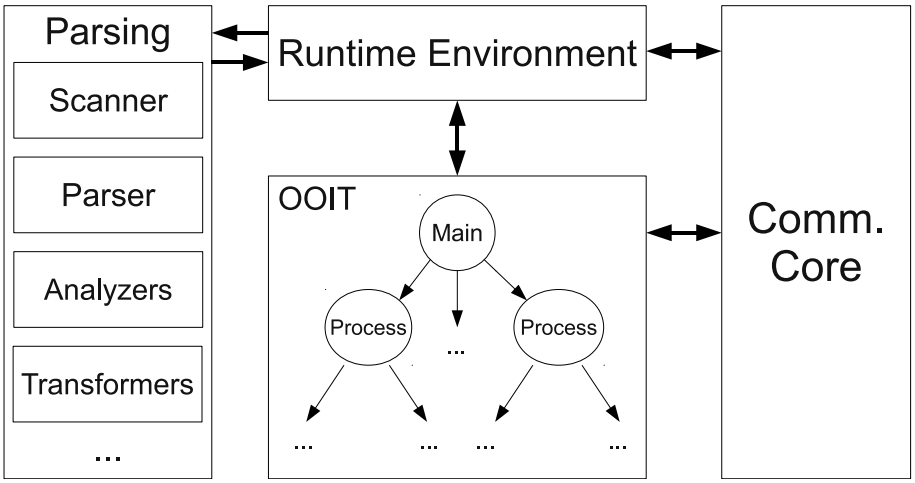

Figure 6.1: A graphical representation of the JOLIE interpreter structure

- *Communication Core*: handles communications, allowing the other components to treat input and output messages abstracting from the underlying communication mediums and protocols.

The interpreter structure is graphically represented in fig. 6.1.

6.1 Interpretation algorithm

The JOLIE interpretation algorithm can be summarized in the following steps:

1. Command line arguments are read, and the components needed for parsing are started;
2. the source code in input is parsed and an Abstract Syntax Tree (AST) is produced;
3. well-formedness, semantic checks and optimizations are performed on the AST;
4. the AST is used in order to generate the OOIT and initialize the Communication Core and Runtime Environment components;

5. the Runtime Environment calls the `run` method of the radix node in the OOIT (which corresponds to the `main` procedure).

6.2 Input parsing and abstract syntax tree

Input source code is tokenized through a scanner (implemented in `jolie.lang.parse.Scanner`), which follows the behaviour of a DFA (Deterministic Finite Automaton). The scanner reads its input through a generic `InputStream`; thus, in principle, it could be used to parse input code from resources that are not files, such as network-backed streams. Comments are automatically filtered out, so that they do not reach the upper level (that of the parser, presented in the following).

Extending the language for supporting new tokens can be done by adding new elements to the `Scanner.TokenType` enumeration, which represents the list of *keywords* in the JOLIE language. The actual implementation for reading the token must be put inside the `Scanner.getToken` method.

The tokens are used by the JOLIE parser (`jolie.lang.parse.OLParser`), implemented as a *recursive descent parser*, which checks if the source code respects the grammar of the language and produces the respective *Abstract Syntax Tree* (AST). Each node of the AST extends class `jolie.lang.parse.ast.OLSyntaxNode`. Class `OLSyntaxNode` offers support for obtaining parsing information, such as the line of source code from which the node has been generated. Moreover, it is instrumental to the implementation of AST analyzers and transformers.

The Abstract Syntax Tree can be analyzed and/or transformed by means of visitors [15]. JOLIE comes equipped with two visitors that are immediately used after the parsing phase. The first one is `OLParseTreeOptimizer`, which transforms the program AST into an optimized version by reducing, when possible, the number of nodes and, in some cases, even by transforming some code in more efficient versions. The second one is `SemanticVerifier`, which performs well-formedness and semantic checks on the parsed program.

6.3 Execution: OOIT and Runtime Environment

After the parsing phase has been performed, the AST is passed to an `OOITBuilder` object. `OOITBuilder` reads the AST and produces the corresponding OOIT.

The *Object-Oriented Interpretation Tree* (OOIT) is an object tree-like data structure that defines the semantics for program execution. Each node of the OOIT implements the `jolie.process.Process` interface. This design has been chosen in order to allow programmers to implement semantic rules, following the foundational studies in SOCK, in a reasonably encapsulated manner. Indeed, each node is responsible for implementing the semantics of a single statement or statement composer, and the implementation of a semantic rule of SOCK is usually contained in a single class that implements `Process`. The definition of interface `Process` follows:

```
public interface Process
{
    public void run()
        throws FaultException, ExitingException;

    public Process clone( TransformationReason reason );

    public boolean isKillable();
}
```

where method `run` is meant to implement the semantics of the process node, method `clone` is used in dynamic (at runtime) transformations of the OOIT and method `isKillable` implements the killable predicate that has been presented in 5.2.

Some process nodes may need to access the state of a session or some other shared data structures. The Runtime Environment component of-

fers methods for accessing these structures without needing to worry about race conditions. Moreover, the Runtime Environment handles parallel execution by means of native threads. Created threads are traceable because they all extend the base `jolie.JolieThread` class. Threads responsible for executing a part of the OOIT extend a subclass of `JolieThread`: `jolie.ExecutionThread`. Execution threads handle the dynamic state of scopes, managing the updates to fault, termination and compensation handlers. Execution threads can be of two kinds: session threads (`jolie.SessionThread`) and parallel threads (`jolie.runtime.ParallelThread`). Session threads are used for handling different sessions and retain a local state for variable values, whereas parallel threads are used for handling parallel composition and refer to their parent session thread for state handling.

6.4 Communications

The OOIT performs communications by exploiting the Communication Core component (`jolie.net.CommCore`). The Communication Core is based upon two main abstraction mechanisms, those of *messages* and *channels*.

A message (`jolie.net.CommMessage`) is formed by the following parts:

- a resource path, which is to be processed by the receiving party as explained in 4.3;
- the name of the operation it is meant for;
- the (structured) data of the message, i.e. its content;
- optionally, if the message is meant to transmit a fault signal, a fault name.

Messages are abstract: they do not contain information about encoding or decoding using specific protocols.

Channels (`jolie.net.CommChannel`) allow for the sending and receiving of messages. As such, a channel is responsible for encoding/decoding a message using the right format and then to send/receive it by means of the right communication medium. Some channels (such as those using in-memory local communications) do not encode/decode data at all, because they use native JOLIE messages. Most channels, however, make use of data protocols. A protocol (`jolie.net.CommProtocol`) specifies how messages should be encoded or decoded. Examples of protocols are SOAP, SODEP or HTTP. Protocols can be used by channels without needing to know their internal details, exploiting a common interface. This allows for the free mixing of protocols with communication mediums and is the key factor in supporting the independency between the `Location` and `Protocol` elements in ports.

Support for new channels and protocols can be added by means of *extensions* (or JOLIE extensions), which are simple JAR (Java Archive) files that can be put inside the `extensions` directory in the JOLIE installation path. A JOLIE extension may contain definitions for new channels and protocols, whose names must be indicated in the JAR Manifest file.

Chapter 7

Programming techniques and examples: using JOLIE

This chapter is devoted to showing some relevant programming techniques that can be used with the JOLIE language in the design and implementation of service-oriented software architectures.

7.1 Interceptors and wrappers

When dealing with large software infrastructures, it is often the case that some services may need to be adapted in order to get integrated with the rest of the architecture. Such adaptation may be related to the behaviour or the deployment definition of the service (or both). Interceptors and wrappers are programming patterns meant to address this kind of issue. The two techniques are similar in their basic concepts but the architectures resulting from their applications are different, and this may affect the interactions with other services, as commented at the end of the section. In the following the term *legacy service* is used to refer to the service that needs to get adapted.

7.1.1 Interceptors

There are two kinds of interceptors: input interceptors and output interceptors. Input interceptors are used to adapt the input interface of the legacy service by intercepting calls for it from other services, whereas output interceptors intercept calls coming out of the output ports of the legacy service. In order to function properly, interceptors require that the rest of the system possesses adequate binding information. In particular, input interceptors usually take the place of the intercepted legacy service. As such, the location of the legacy service needs to be changed to a new one, known by the input interceptor, whereas the input interceptor takes the original location of the legacy service. Output interceptors, instead, are usually meant to be used only by the legacy service and not by the rest of the service-oriented architecture. Because of this the location of the legacy service does not change, but one must perform rebinding on it so to make it use the output interceptor.

Input interceptors

Deployment adaptation with an input interceptor can be easily addressed by means of aggregation. This is obtained by exposing an input port with different deployment information and by making it aggregating an output port towards the legacy service. The following is an example that adapts a legacy service by exposing its functionalities through the SOAP protocol, in order to make it usable by Web Services. The code for the output port definition and the main procedure are not reported: the first one can be of any kind, because aggregation is independent from the used communication medium and protocol, whereas the second one is not relevant because the behaviour of the interceptor does not play any role.

```
outputPort LegacyService { ... }

inputPort InterceptorInput {
Location: "socket://localhost:80/"
```

```
Protocol: soap
Aggregates: LegacyService
}

main { ... }
```

Behavioural adaptation exploits the fact that aggregation gives precedence to the interface of the aggregator. In this case, the aggregator exposes a subset of the aggregated interface and implements it by itself. In order to exemplify, let us consider a simple calculator service which exposes the following interface:

```
interface CalculatorInterface {
OneWay:
    shutdown(void)
RequestResponse:
    sum(SumRequest)(int)
}
```

Integrating this service with some auditing mechanism may require, e.g., to log each received call for the `sum` operation. This can be simply obtained through the following interceptor:

```
execution { concurrent }

interface CalculatorInterface {
OneWay:
    shutdown(void)
RequestResponse:
    sum(SumRequest)(int)
}

interface SumInterface {
RequestResponse:
```

```
    sum(SumRequest)(int)
}

outputPort Calculator {
Location: Location_Calculator
Protocol: sodep
Interfaces: CalculatorInterface
}

outputPort Logger { ... }

inputPort InterceptorInput {
Location: "socket://localhost:80/"
Protocol: soap
Interfaces: SumInterface
Aggregates: Calculator
}

main
{
    sum( request )( response ) {
        sum@Calculator( request )( response );
        log@Logger( "Operation sum has been called" )
    }
}
```

The interceptor implements operation `sum` which, after calling calculator on the same operation, performs the additional logging.

Following the same reasoning, one could also add new operations to the exposed input port or hide some operations of the aggregated service. Adding an operation can be easily done by exposing an additional interface in the input port of the aggregator or by aggregating multiple output ports. Hiding

an operation, instead, is obtained by modifying the interface of the aggregated output port, i.e. by removing the operation of interest from it.

Output interceptors

Output interceptors capture outgoing calls issued by the legacy service and then trigger some actions. In order to introduce this technique, let us consider an example in which a legacy service relies on a mail server for sending E-Mails. The objective is to perform some additional action, e.g. saving a backup copy, whenever the legacy service uses the `sendMail` operation of the mail server. Doing this with an input interceptor on the legacy service could be hard because it is not known, a priori, when the execution of one of its operations will cause an invocation for the `sendMail` operation. Instead, one could use an output interceptor that is responsible for intercepting calls to `sendMail` and performing the backup action:

```
execution { concurrent }

interface MailServerInterface { ... }

interface SendMailInterface {
RequestResponse:
    sendMail(SendMailRequest)(void)
}

outputPort MailServer {
Location: Location_MailServer
Protocol: soap
Interfaces: MailServerInterface
}

outputPort BackupServer { ... }
```

```
inputPort InterceptorInput {
Location: "socket://localhost:9000/"
Protocol: soap
Interfaces: SendMailInterface
Aggregates: MailServer
}

main
{
    sendMail( request )( response ) {
        sendMail@MailServer( request )( response )
        |
        backup@BackupServer( request )
    }
}
```

7.1.2 Wrappers

While interceptors execute as siblings of the legacy service, a *wrapper* executes a legacy service (the *wrapped service*) as an inner service by means of embedding. Wrapping can be applied in all the scenarios considered above with interceptors, and in a very similar way. Indeed, the only difference between the two approaches lies on the resulting architecture. This fact can be relevant: intercepted legacy services are still reachable by the rest of the service system if some other service knows its new location, whereas one can use the `local` communication medium in the case of wrapping in order to completely isolate the legacy service. Moreover, the wrapper and the wrapped services can be treated as a single unit: if the execution of the wrapper gets terminated, the wrapped service is automatically terminated, too. This is not the case with interceptors, where one must take care of terminating also all the related interceptors when a legacy service is stopped.

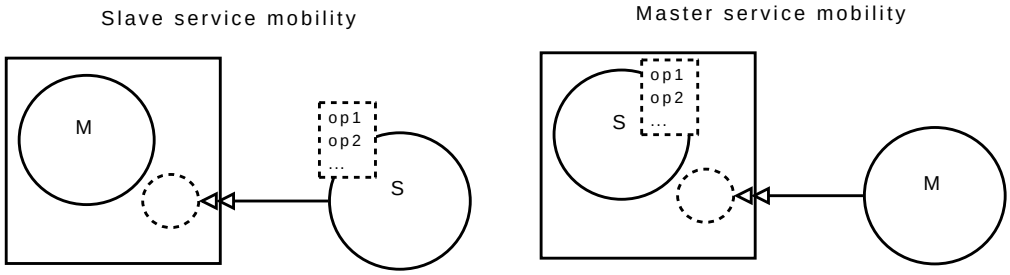

Figure 7.1: Slave service mobility and Master service mobility.

7.2 Service mobility patterns

Service mobility is a direct application of dynamic embedding: service definitions are transmitted over the network and embedded by the receiver. Service mobility can be of two different kinds: slave service mobility and master service mobility. A *slave* service provides simple functionalities; a *master* service, instead, makes use of multiple slave services in its workflow. In the *slave service mobility* pattern slave services are transmitted to and embedded by a master service, whereas in the *master service mobility* the master service is moved and embedded into the same execution engine of the slaves. In the following both patterns are explained by referring to Fig. 7.1.

- **Slave service mobility.** Consider the case in which a service M (the *master service*) defines a workflow that is dependent on some functionalities that cannot be provided statically before execution time. M, instead, needs to obtain these functionalities at runtime and to use them. In order for this to work, M must define an appropriate output port for the functionalities it is looking for. Then, M asks a service repository for downloading the functionalities it needs. The repository sends a service S to M, and M dynamically embeds S. M has now access to the functionalities offered by S, the *slave service*, and exploits them to complete its workflow.

- **Master service mobility.** Consider the case in which a service S (the *slave service*) possesses the functionalities that are needed for the

execution of a workflow, but the workflow cannot be provided statically before execution time. S needs to obtain the workflow at runtime and to execute it, ensuring that the workflow makes use of the functionalities provided by S. Such a pattern is suitable, e.g., for implementing the SENSORIA automotive scenario [43], where a car experiments a failure and starts a recovery workflow for booking some services such as the garage, the car rental and the truck one. A prototype of the scenario has been implemented in JOLIE with a slave service on the car and the master workflow which is downloaded from the car factory assistance service. This way the recovery workflow can be changed and maintained by the car assistance service without updating all the car software periodically and, at the same time, it guarantees transaction security by isolating some functionalities, such as the bank payment, into the slave services of the car. In this case the downloaded workflow is able to search for all the services it needs and it relies upon the slave service car functionalities for the payment.

7.3 SoS: service of services pattern

The SoS pattern exploits both dynamic embedding and dynamic redirecting. A service is embedded at run-time if a client performs a resource request to the master service. In this case the master service embeds the requested service by downloading it from a repository and makes it available to the client with a private resource name. The client will now be able to perform invocations to its own resource by addressing its requests to the resource name it has received. The main advantage of this approach is that one can provide an entire service as a resource to a specific client instead of a single session of a service. This pattern has been used for implementing a prototype of the SENSORIA finance case study [43], which models a finance institute where several employees works on the same data. The SoS pattern is there exploited for loading a service for each employee, which maintain its

private data and, at the same time, can offer a set of functionalities. The main advantage in this approach is that each functionality offered to the employee is able to open a session on its own service thus obtaining a private complex resource made of a set of service sessions.

7.3.1 MetaService

MetaService, available in the JOLIE standard library, is a service that is specifically designed for generalizing the application of the SoS pattern. As such, MetaService offers an interface for performing dynamic embedding and setting new redirections. For the sake of clarity here a simplified, yet functional, version of MetaService is reported. The complete version can be found in the JOLIE standard library [24].

The basic interface of MetaService follows:

```
type MetaData:void { ? }

type ServiceRecord:void {
    .resourceName:string
    .metadata:MetaData
}

type ServiceList:void {
    .service[0,*]:ServiceRecord
}

type LoadEmbeddedJolieServiceRequest:void {
    .resourceName:string
    .definition:string
    .metadata:MetaData
}

interface MetaServiceAdministration {
```

```
RequestResponse:
    getServices(void)(ServiceList),
    loadEmbeddedService
        (LoadEmbeddedJolieServiceRequest)(void)
        throws EmbeddingFault,
    unloadEmbeddedService(string)(void)
}
```

The interface is composed by two operations:

- `getServices`: it returns a list of the currently loaded services;
- `loadEmbeddedService`: it receives a service definition to load, possibly equipped with descriptive metadata, and the resource name to be given to the loaded service;
- `unloadEmbeddedService`: it stops the service currently executing under the passed resource name.

MetaService executes its sessions sequentially, because each operation invocation causes accesses to global data structures in order for its response to be computed. The main part of MetaService follows:

```
execution { sequential }

inputPort MetaService {
Location: MetaServiceLocation
Protocol: MetaServiceProtocol
Interfaces: MetaServiceInterface
}

main
{
    [ getServices()( response ) {
        i = 0;
```

```
        foreach( s : global.services ) {
            if (
            is_defined( global.services.(s).metadata )
            ) {
                response.service[i].metadata <<
                    global.services.(s).metadata
            };
            response.service[i].resourceName = s;
            i++
        }
    } ] { nullProcess }

    [ loadEmbeddedJolieService( request )() {
        scope( EmbedScope ) {
            install(
                RuntimeException =>
                    throw( EmbeddingFault )
            );
            file.filename =
                request.resourceName + ".ol";
            file.content = request.definition;
            writeFile@File( file )();
            with( embedInfo ) {
                .type = "Jolie";
                .filepath = file.filename
            };
            loadEmbeddedService@Runtime
                ( embedInfo )( handle );

            with( port ) {
                .name = request.resourceName;
```

```
                .location = handle
            };
            setOutputPort@Runtime( port )();
            redirection.outputPortName =
                redirection.resourceName;
            redirection.inputPortName =
                "MetaService";
            setRedirection@Runtime( redirection )();
            global.services.
                (redirection.resourceName).metadata
                << request.metadata
        }
    } ] { nullProcess }

    [ unloadEmbeddedService( resourceName )() {
        if (
        is_defined( global.services.(resourceName) )
        ) {
            service -> services.(resourceName);
            r.resourceName = request;
            r.inputPortName = "MetaService";
            removeRedirection@Runtime( r )();
            removeOutputPort@Runtime( request )();
            callExit@Runtime
                ( service.privates.handle )()
            undef( global.services.(resourceName) )
        }
    } ] { nullProcess }
}
```

The three operations are put in a nondeterministic input choice. This, coupled with the sequential execution of sessions, ensures that only one of these

operation bodies is executed at a time. The `global.services` tree stores information about each loaded service. The most interesting part is the body of operation `loadEmbeddedService`. There, the received definition is first written into a file and then loaded as an embedded service by means of the `Runtime` service (provided by the JOLIE standard library). After that, an output port bound to the newly created service is created, and then used to dynamically set up a redirection through the `setRedirection` operation. Finally, the global `services` data structure is updated. Intuitively, operation `unloadEmbeddedService` reverts the situation created by a call to `loadEmbeddedService`.

7.3.2 Example

Let us consider a simple example, in which a company offers to customers the possibility to execute services by means of service mobility. Each customer has a certain amount of allowed time: each loaded service can not execute for more time than what the customer is allowed for, and when the service terminates the allowed time gets proportionally decreased. Such a service could be offered by means of an orchestrator that composes MetaService in order to load and unload the services sent by the customers. The following is a possible prototype where, for the sake of brevity, only the relevant parts of the behavioural and deployment definitions are reported:

```
cset { sid }

execution { concurrent }

main
{
    login( request )( sid ) {
        authenticate@AccountManager
            ( request )( account );
        synchronized( SessionCreation ) {
```

```
            sid = global.sid++
        }
    };
    startService( startRequest )() {
        loadEmbeddedService@MetaService
            ( startRequest )();
        setNextTimeout@Time( account.allowedTime )
    };
    [ timeout() ] { nullProcess }
    [ stopService( sid ) ] { nullProcess };
    {
    unloadEmbeddedService@MetaService
        ( startRequest.resourceName )()
    |
    updateAllowedTime@AccountManager( account )()
    }
}
```

The service is executed in a concurrent modality, so to allow for multiple client sessions. First, the customer is required to login. An `AccountManager` service is composed and is responsible to handle customer accounts. After the customer has been successfully authenticated, a session id `sid` is sent to the customer. The session id is in the service correlation set, so the customer can use it to refer to the created session later. The `startService` operation is then made available, which can be called in order to start a new service; the latter is loaded by composing MetaService. When the service is successfully embedded, the `Time` service is used in order to handle a timeout that is set to the allowed time of the customer. This timeout is used in the following nondeterministic choice, where either the timeout occurs or the `stopService` operation gets called first. In both cases the service gets unloaded and, concurrently, the account allowed time gets updated.

Conclusions

The JOLIE language represents, to the best of the author's knowledge, the first attempt at the implementation of a full-fledged programming language that is entirely based upon the service-oriented programming paradigm. The distinguishing difference between JOLIE and other industrial service-oriented languages, such as BPEL, is that one of the aims of JOLIE is to be ubiquitous, targeting various application environments based upon different communication technologies. The effort put in making it a general approach to service orchestration is especially visible from the level of abstraction that it offers in communications. Furthermore, the fact that the JOLIE interpreter is lightweight extends its applicability and opened up the possibility to experiment with new levels of granularity in service-oriented architectures; perhaps the most important consequence of this experimentation is that it brought the intuition for the creation of the architecture-related mechanisms described in Chapter 4: *aggregation*, *embedding* and *redirection*. The mechanisms and programming techniques that have been developed in JOLIE have already been of inspiration for other works within the scopes of evolvability [35] and adaptability [26]. Moreover, the presented dynamic error handling mechanisms, which allow for error handlers to be updated at runtime, fit nicely the dynamic nature of long running transactions in service systems and has been proven to possess interesting expressiveness results [28].

This thesis, which has been partially conducted under the scope of European Project SENSORIA [43], has already been validated during its development by means of academic publications and industrial application; a

brief survey will follow.

The separation between the behavioural and deployment parts in JOLIE programs is reported in [34]. Dynamic fault and compensation handling has been first studied from a foundational point of view in [18, 19]. Its implementation and implications in the JOLIE language are presented in [33]. JOLIE has also been used to implement a distributed architecture for the management of virtual machines [7]. Finally, and perhaps most importantly, the progresses on SOCK and JOLIE are continuously laying the foundations for studying the peculiarities of the service-oriented programming paradigm. The first results of this study have been reported in [21].

On the industrial side JOLIE is the reference development language of italianaSoftware s.r.l. [23], an IT company offering service-oriented solutions. Among the developed software applications one can cite **CentralWatcher**, a software for the integration of phone switches with service-oriented systems where the embedding mechanism plays a key role, and **Guide One Page** [41], a Web 2.0 portal for tourism whose backend is entirely supported by JOLIE (even the server that communicates with the web browser, by means of the JOLIE HTTP protocol). Other applications are QtJolie, a C++ library for handling communications with JOLIE services developed in the scope of the KDE project [25], and **Vision**, a software for handling the distributed synchronization of presentations that can be found in the JOLIE public source code repository.

Related work Other languages for service programming equipped with a formal semantics and in topic with this thesis are Blite [30] and PiDuce [11]. Blite focuses on supporting Web Services and BPEL, while JOLIE represents a more general approach to service-oriented programming, considering also service-oriented technologies that are not based upon XML, legacy applications and mechanisms that are not present in BPEL specifications. Moreover, Blite compiles its programs to BPEL code; this brings the risk to compromise the advantage given by the formality of its semantics, because the actual

execution is done by BPEL engines, which do not come with formal specifications. PiDuce, instead, proposes a model inspired to the π-calculus [42], based on channels and not on correlation sets. As such, its connection towards the Web Service technology is less intuitive. The language does not offer mechanisms for code mobility, where JOLIE features the transmission and execution of service definitions.

Future work

Future work comprises both foundational and technological developments.

The SOCK process calculus will be updated in order to better model the advanced features offered by JOLIE w.r.t. state handling and architectural composition. This work will be succeeded by the development of a theory for manipulating JOLIE interface definitions, in order to study more powerful operators for their composition and, in turn, enhance the aggregation mechanism. This step would open up the possibility to create general interceptors (or wrappers) that modify the interface of the intercepted (or wrapped) service without needing to know its entire interface definition. For instance, one could create a standard service for adding authentication capabilities to a legacy service.

Subsequently, the preliminary work that has been conducted for studying formal relations between orchestration and choreography in [27] will be extended so to create a framework for choreography-driven programming based upon JOLIE. Such a framework will be coupled with development tools and an Integrated Development Environment (IDE) based upon formal notions of End-Point Projection (EPP) [10, 22, 27] and Global View Extraction (GVE). EPP and GVE allow, respectively, for the automatic translation of a choreography definition into skeleton code that defines the necessary communications for each end-point and its reverse, i.e. mapping end-point code into a choreography definition. EPP and GVE provide what is best known as round-trip engineering: existing source code can be abstracted and converted

into a specification, subjected to software engineering methods and then converted back. Establishing relations such as EPP and GVE in a sound manner is important so to ensure that system properties are safely preserved when going from choreography to orchestration and vice versa. As such, the formal background of the JOLIE language makes the latter a very good candidate for being a target language for projection. The high grade of granularity that embedding introduces makes it interesting to consider relations such as that presented in [9], where the authors present a notion of conformance between choreography and orchestration in which a choreography role can be mapped to multiple orchestrators. This is relevant because in practice it is often the case that a JOLIE service makes use of several small sub-services in order to function. Moreover, dynamic error handling will surely need to be introduced in the choreography language connected to JOLIE. This will cause the need to update the current notions of choreography correctness.

EPP can be mixed with concepts such as session types [22] and service contracts [12] in order to achieve the automatic composition of the services needed for implementing the global specification [8]. Indeed, one can define a notion of EPP that maps a choreography to a set of session types or contracts and then make use of some registry for performing a lookup of services available in the network that would be conformant to them. This fact motivates an investigation for the adaptation of these techniques to JOLIE. Such an investigation will lead to the development of a language for expressing JOLIE behavioural types (or contracts).

Ultimately, the exposed service mobility features will be further developed in order to offer more powerful adaptation mechanisms. This will lead to the implementation of native primitives for dynamic embedding and it will also need to be considered in the design of the aforementioned choreography language, under the form of services as first-class values.

Bibliography

[1] Google Web Toolkit. http://code.google.com/webtoolkit/.

[2] OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.

[3] SODEP: Simple Operation Data Exchange Protocol. http://www.jolie-lang.org/wiki.php?page=Sodep.

[4] Web Services Description Language (WSDL) Version 2.0. http://www.w3.org/TR/wsdl20/.

[5] *Web Services Glossary*. http://www.w3.org/TR/ws-gloss/.

[6] XML-RPC. http://www.xmlrpc.com/.

[7] P. Anedda, M. Gaggero, S. Manca, O. Schiaratura, S. Leo, F. Montesi, and G. Zanetti. A general service oriented approach for managing virtual machines allocation. In *Proceedings of ACM Symposium on Applied Computing (SAC), 2009*, pages 2154–2161, 2009.

[8] Mario Bravetti, Ivan Lanese, and Gianluigi Zavattaro. Contract-Driven Implementation of Choreographies. In *Proceedings of TGC 2008*, pages 1–18, 2008.

[9] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration conformance for system design. In *Proceedings of 8th International Conference on Coordination Models and Languages*

(COORDINATION 2006), volume 4038 of *Lecture Notes in Computer Science*, pages 63–81, 2006.

[10] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *Proceedings of ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2007.

[11] Samuele Carpineti, Cosimo Laneve, and Luca Padovani. PiDuce - A project for experimenting Web services technologies. *Science of Computer Programming*, 74(10):777–811, 2009.

[12] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for Web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.

[13] Elvis Ciotti. *Implementazione di un sistema di tipi per JOLIE (Implementation of a type system for JOLIE.* Msc. thesis, Department of Computer Science, University of Bologna.

[14] Free Software Foundation (FSF). GNU Lesser General Public License. `http://www.gnu.org/licenses/lgpl.html`.

[15] R. Johnson J. Vlissides G. Erich, R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[16] C. Guidi and R. Lucchi. Programming service oriented applications. In *UBLCS-2008-11, Technical Report, Department of Computer Science, University of Bologna*, 2008. `http://www.cs.unibo.it/pub/TR/UBLCS/2008/2008-11.pdf`.

[17] Claudio Guidi. *Formalizing languages for Service Oriented Computing.* PhD. thesis, Department of Computer Science, University of Bologna, 2007. http://www.cs.unibo.it/pub/TR/UBLCS/2007/2007-07.pdf.

[18] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. On the interplay between fault handling and request-response service invocations. In *Proceedings of ACSD 2008*, pages 190–198, 2008.

[19] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic Error Handling in Service Oriented Applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.

[20] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proceedings of ICSOC 2006*, pages 327–338, 2006.

[21] Claudio Guidi and Fabrizio Montesi. Reasoning About a Service-oriented Programming Paradigm. In *Proceedings of YR-SOC 2009*, pages 67–81, 2009.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL 2008*, pages 273–284, 2008.

[23] italianaSoftware s.r.l. italianaSoftware. `http://www.italianasoftware.com/`.

[24] JOLIE. JOLIE: Java Orchestration Language Interpreter Engine. `http://www.jolie-lang.org/`.

[25] KDE. The K Desktop Environment. `http://www.kde.com/`.

[26] Ivan Lanese, Antonio Bucchiarone, and Fabrizio Montesi. A Framework for Rule-based Dynamic Adaptation. In *Proceedings of TGC 2010, 5th International Symposium on Trustworty Global Computing*, Lecture Notes in Computer Science. SV, 2010.

[27] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *Proceedings of Sixth IEEE International Conferences on Soft-*

ware Engineering and Formal Methods (SEFM 2008), pages 323–332. IEEE Computer Society, 2008.

[28] Ivan Lanese, Catia Vaz, and Carla Ferreira. On the Expressive Power of Primitives for Compensation Handling. In *Proceedings of ESOP 2010*, Lecture Notes in Computer Science. SV, 2009. To appear.

[29] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[30] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Formal Account of WS-BPEL. In *Proceedings of COORDINATION 2008*, pages 199–215, 2008.

[31] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.

[32] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[33] Fabrizio Montesi, Claudio Guidi, Ivan Lanese, and Gianluigi Zavattaro. Dynamic Fault Handling Mechanisms for Service-Oriented Applications. In *Proceedings of ECOWS 2008*, pages 225–234, 2008.

[34] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing Services with JOLIE. In *Proceedings of ECOWS 2007*, pages 13–22, 2007.

[35] Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In *Proceedings of Fifth Symposium on Trustworthy Global Computing (TGC 2010)*, 2010. To appear.

[36] OASIS. UDDI Specifications. `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`.

[37] OASIS. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/`.

[38] OASIS. Web Services Coordination. `http://docs.oasis-open.org/ws-tx/wscoor/2006/06`.

[39] OASIS. Web Services Security. `http://www.oasis-open.org/specs/index.php#wssv1.1`.

[40] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[41] Guide One Page. Guide One Page. `http://www.guideonepage.com/`.

[42] Davide Sangiorgi and David Walker. *The π-Calculus. A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[43] SENSORIA. Software Engineering for Service-Oriented Overlay Computers. `http://www.sensoria-ist.eu/`.

[44] M. Viroli. Towards a Formal Foundation to Orchestration Languages. In M. Bravetti and G. Zavattaro, editors, *Proc. of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*, volume 105 of *ENTCS*. Elsevier, 2004.

[45] World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[46] World Wide Web Consortium (W3C). HTTP - Hypertext Transfer Protocol. `http://www.w3.org/Protocols/`.

[47] World Wide Web Consortium (W3C). SOAP Specifications. http://www.w3.org/TR/soap/.

[48] World Wide Web Consortium (W3C). Web Services Addressing. http://www.w3.org/TR/ws-addr-core/.

[49] World Wide Web Consortium (W3C). Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[50] World Wide Web Consortium (W3C). Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

[51] World Wide Web Consortium (W3C). Web Services Description Language. http://www.w3.org/TR/wsdl.

Acknowledgments

This thesis marks the end of one of my most intense and fulfilling periods, both professionally and personally speaking. While writing this last page I feel satisfied and I look forward to turn it and start with the next chapter of my life; still, I am really enjoying this moment and I want to thank the people that made this possible.

My first words of thank go to my family.
I thank my wife, Maja, for her immense personal and intellectual support in these years; for her great honesty and brilliancy, through which she enriches me. Maja, you are an exceptional person. Thank you for our wonderful relationship.
I thank my parents, Paolo and Irene, for raising me by giving me confidence and freedom, and for all the love and trust I have received. I have no words to express how precious these gifts are, and how much they helped me along my way.
I thank my brother and sister, Marco and Dania, for being so joyful and transmitting me the feeling that we can do anything. The support we share for each other is a treasure to me.

Next, I thank all the people I worked with at the University of Bologna; working with them has been a pleasure and a priviledge.
Thanks to Dr. Claudio Guidi for our endless and exciting discussions about the service-oriented paradigm. Our journey still continues within italiana-Software.
I thank my supervisor, Prof. Gianluigi Zavattaro, for being so knowledgeable

and open to discussion, and for all the good advices he gave me.
I also thank Dr. Ivan Lanese for the many interesting technical debates, occasionally followed by refreshing beer.
I extend my gratitude to Prof. Davide Sangiorgi, for his positive support to the JOLIE project and our inspiring discussions about foundations for distributed systems.
Finally, I wish to thank Prof. Roberto Gorrieri for giving me the opportunity of working in EU Project SENSORIA, which has been the starting point for all my research.

Special thanks to all my friends at the Department of Computer Science in Bologna, for all the lunches shared together and for making my days feel lighter.
I must also thank the administrative staff of the same department; it is, indeed, composed by great people.

I wish to thank also all my old friends from my hometown, for keeping in touch while I were so busy, and all the friends that I made during these last years for the fun we shared together.