

UNIVERSITÀ DEGLI STUDI DI UDINE

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

CORSO DI LAUREA IN TECNOLOGIE WEB E MULTIMEDIALI

BACHELOR THESIS

Design and implementation of a Language Server for the Jolie programming language

CANDIDATE

Eros Fabrici

SUPERVISOR

Prof. Marino Miculan

Academic Year 2018-2019

INSTITUTE CONTACTS

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Università degli Studi di Udine

Via delle Scienze, 206

33100 Udine — Italia

+39 0432 558400

<http://www.dimi.uniud.it/>

Al mio cane,
per avermi ascoltato mentre ripassavo le lezioni.

Acknowledgements

Sed vel lorem a arcu faucibus aliquet eu semper tortor. Aliquam dolor lacus, semper vitae ligula sed, blandit iaculis leo. Nam pharetra lobortis leo nec auctor. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Fusce ac risus pulvinar, congue eros non, interdum metus. Mauris tincidunt neque et aliquam imperdiet. Aenean ac tellus id nibh pellentesque pulvinar ut eu lacus. Proin tempor facilisis tortor, et hendrerit purus commodo laoreet. Quisque sed augue id ligula consectetur adipiscing. Vestibulum libero metus, lacinia ac vestibulum eu, varius non arcu. Nam et gravida velit.

Abstract

Nunc ac dignissim ipsum, quis pulvinar elit. Mauris congue nec leo ornare lobortis. Nulla hendrerit pretium diam nec lobortis. Nullam aliquam laoreet nisl, sit amet facilisis lectus accumsan ut. Duis et elit hendrerit metus venenatis condimentum. Integer id eros molestie, interdum leo sit amet, aliquet metus. Integer fermentum tristique magna, vel luctus neque rhoncus vel. Ut hendrerit et quam et semper. Mauris egestas, odio sed aliquet luctus, magna orci euismod odio, vitae lacinia tellus tellus non lectus. Aliquam urna neque, porta et mattis aliquam, congue sit amet lorem. In ultrices augue sit amet ante vehicula, vitae rhoncus turpis auctor. Donec porta scelerisque eros, at mollis enim imperdiet ut.

Contents

Contents	ix
1 Introduction	3
2 Jolie and the Language Server Protocol	5
2.1 Jolie: Java Orchestration Language Interpreter Engine	5
2.1.1 Behavior	6
2.1.2 Deployment	9
2.2 Language Server Protocol	15
2.2.1 How LSP works	15
3 Analysis of the problem	19
3.1 Language Server Protocol features	19
3.2 Functional Requirements	19
3.3 Non-functional requirements	20
3.3.1 Complying with the Language Server Protocol specification	20
3.3.2 Distributed	21
3.3.3 Modular	21
4 System architecture	23
4.1 Architecture	23
4.2 Services	25
4.2.1 Main service	25
4.2.2 Text Document service	26
4.2.3 Workspace service	26
4.2.4 Utils service	26
4.2.5 Forwarder service	27
5 Implementation	29
5.1 Complying with LSP JSON-RPC specifications	30
5.1.1 LSP message and parser	30
5.1.2 Aliasing LSP method names to Jolie operation names	31
5.2 Server implementation	33
5.2.1 Operation types and interfaces	33
5.2.2 Main service	34
5.2.3 Utils service	36
5.2.4 Text Document service	36
6 Validation	39
6.1 Met requirements (screenshots)	39
6.2 Unmet requirements	39
7 Conclusions: review, what needs to be done	41

CONTENTS	1
I Appendici	43
A Altro capitolo	45
Bibliography	49

1

Introduction

Jolie and the Language Server Protocol

2.1 Jolie: Java Orchestration Language Interpreter Engine

Service-Oriented Computing (SOC) is a design methodology that focuses on the composition of autonomous entities in a system, called services. SOC abstracts from the implementation details of services by imposing a standard communication mechanism between the entities in an SOA (Service-Oriented Architecture)[1]. From the perspective the aforementioned methodologies, it is possible to identify a separation between *behavioral* and *architectural* composition of services. The first denotes a series of interactions to be performed in order to reach a goal, while the second deals with the topological structure of a SOA, managing its execution and integration. The lack of an homogeneous solution between the *behavioral* and the *architectural* composition of services require ad-hoc interventions in order to integrate them, which will make the entire architecture less modular and difficult to maintain.

Jolie was created with the intention to fill the aforementioned gap. This language permits to define services, their behavioral composition, supporting different communication technologies, and their organization inside a SOA.

A Jolie program defines a service, which is composed by two parts: the *behavior* and the *deployment*. The first defines the functionalities of the service, including primitives like communication and computation constructs. These does not define *how* the communication is supported, they abstractly refer to *communication ports* which are assumed to be correctly defined in the deployment part. For example, a behavioral primitive may express the action "ask the bank to show the actual balance", without knowing how to reach the "bank" service and which communication protocol it uses. These last two are handled by the deployment, which permits to define the *location* and the *protocol* of the bank service. Furthermore, the deployment permits the usage of architectural primitives for defining the architecture of a SOA. The syntax of a Jolie program is defined as the following:

$$Program ::= D \text{ \textcolor{red}{main}} \{ B \}$$

where D is the Deployment and B is the Behavior. The **main** procedure is the execution entry point.

2.1.1 Behavior

Jolie syntax is a combination of the message passing and the imperative programming styles. Following a selection of the Jolie behavioral syntax:

$$\begin{aligned}
B \quad &::= \eta \quad (\text{input}) \\
&| \bar{\eta} \quad (\text{output}) \\
&| \textcolor{red}{if}(e) B_1 [\text{else } B_2] \quad (\text{condition}) \\
&| \textcolor{red}{while}(e) B \quad (\text{while}) \\
&| \textcolor{red}{for}(\text{init}, \text{cond}, \text{incr}) B \quad (\text{forLoop}) \\
&| \textcolor{red}{for}(x \textcolor{red}{in } y) B \quad (\text{arrayIteration}) \\
&| B_1 ; B_2 \quad (\text{sequential}) \\
&| B_1 | B_2 \quad (\text{parallel}) \\
&| \{ B \} \quad (\text{block}) \\
&| x = e \quad (\text{assign}) \\
&| a - > e \quad (\text{alias}) \\
&| x << y \quad (\text{deepCopy}) \\
&| [\eta_1] \{ B_1 \} \dots [\eta_n] \{ B_n \} \quad (\text{inputChoice}) \\
&| \textcolor{red}{scope}(s) \{ B \} \quad (\text{scope}) \\
&| \textcolor{red}{install}(h_1 => B_1, \dots, h_n => B_n) \quad (\text{install}) \\
&| \textcolor{red}{throw}(f[, x]) \quad (\text{throw})
\end{aligned}$$

where

$$\begin{aligned}
\eta \quad &::= \text{opName}(x) \quad (\text{OneWay}) \\
&| \text{opName}(x)(e) \{ B \} \quad (\text{RequestResponse})
\end{aligned}$$

and

$$\begin{aligned}
\bar{\eta} \quad &::= \text{opName@OP}(e) \quad (\text{notification}) \\
&| \text{opName@OP}(e)(x) \quad (\text{SolcitResponse})
\end{aligned}$$

Communications

Communications are implemented through *Rules* (*input*), (*output*) and (*inputChoice*). Input η can be either a *OneWay* or a *RequestResponse*. The first receives a message for the operation *opName* and saves its content in variable x , while the second receives a message for operation *opName* in variable x ,

executes behavior B and sends the value of the evaluation of the expression e to the invoker. (*notification*) and (*SolcitResponse*) implement the output towards the input . The first sends a message with the evaluation of e , the second does the same thing and waits for the response from the invoked service which is stored in the var \mathbf{x} . OP denotes the **output port name** that references an **outputPort**. The latter is specified in the deployment definition and it contains all the necessary information in order to contact the target service. Lastly, the (*inputChoice*) supports the receiving of a message for any of the inputs. When a message for the input η_i is received, all the remaining branches are deactivated and η_i is executed. Afterwards, B_i is executed. The interpreter enforces that $\eta_i \neq \eta_j, \forall i, j \in [1, n] | i \neq j$, which means that there cannot be two inputs with the same operation.

Statement compositions, such that (*condition*), (*while*), (*forLoop*), (*arrayIteration*) and (*block*), resembles the ones used in classic imperative styles. Rule *sequential* sequentially compose the two behaviors: when B_1 is terminated, B_2 can start its execution (remark: the separator $;$ is now optional). On the contrary, (*parallel*) permits the parallel execution of the two behaviors.

Assignment and aliasing: (*assign*) evaluates the expression e and assigns its value to the variable \mathbf{x} . (*alias*) permits to define a variable a that points to x variable path.

Handling data Jolie is dynamically-typed meaning that there is no necessity of type declarations and that the type of a variable value is checked at runtime. The language handles the following basic data types:

- **bool**: booleans;
- **int**: integers;
- **long**: long integers (with l or L suffix);
- **double**: double-precision float;
- **string**: strings;
- **raw**: byte arrays (supported for data-passing purposes);
- **void**: the empty type.

Jolie supports basic arithmetic and logic operators like in Java and also pre-/post-increment and pre-/post-decrement operators. Furthermore, it is possible to cast variables to other types with the following casting functions: `bool()`, `int()`, `long()`, `double()` and `string()`. For instance:

```

1  d = "1.5"
2  n = double( d ) // n = 1.5
3  n = int( n ) // n = 1

```

A variable type can be checked at run-time with the `instanceof` operator, whose syntax is:

expression `instanceof` (*native_type* | *custom_type*)

Variables are undefined until they are assigned a value. Until then, their state is set to **undefined** (equivalent to `NULL`, `null` or `nil` in other languages). The predicate `is_defined` is used for checking is a

variable is defined or not. Following the syntax:

is_defined(variable_name)

An example:

```

1  x = 2
2  if ( is_defined( x ) ) {
3    println@Console( "Variable x is defined" )
4  }
```

Arrays in Jolie are handled dynamically and can be accessed using the `[]` operator. To be more precise, in Jolie **every** variable is a dynamic array. So for example this assignment

```

1  x = 7
```

is equivalent to the following

```

1  x[0] = 7
```

thus managing complex data is easier. To compute the size of an array Jolie provides the size operator `#`. Together with this operator, the aforementioned *forLoop* can be used to traverse an array. Alternatively, (*arrayIteration*) has the same purpose. The example below shows the usage of the mentioned constructs.

```

1  x[0] = 1
2  x[1] = 2
3  x[2] = 3
4  println@Console( #x )() // prints 3
5
6  for( i = 0, i < #x, i++ ) {
7    println@Console( x[i] )()
8  }
9  //which is equivalent to
10 for( el in x ) {
11   println@Console( el )()
12 }
```

Data Structures

Data structures in Jolie are tree-like, similarly to XML or JSON data trees. The operator `.` is used to traverse a data structure. Following an example.

```

1  doctor.name = "Marco"
2  doctor.surname = "Rossi"
3  doctor.age = 45
4  doctor.specialization[0] = "Cardiology"
5  doctor.specialization[1] = "General Surgery"
```

The above data tree is catheterized by `doctor` as root with `void` type and by three children: `name`, `surname` and `specialization` which respectively have the types `string`, `string`, `int` and `string`. The

`undef` operator is used to erase an entire structure. Syntax:

$$\text{undef}(\text{variable_name})$$

The (*deepCopy*) operator copies the entire structure of a tree y into a tree x . Jolie at run-time explores the tree y node-wise and for all initialized sub-nodes in y , it assigns the value of each of them to the corresponding sub-node rooted in x . This means that if x already had sub-nodes initialized that match to some ones rooted in y , $x \ll y$ will overwrite them, leaving all the others initialized nodes of x unaffected. This operator can be used also as syntax sugar for creating a data structure. Basing on the previous example of the doctor data structure, in can be re-written as in the following example.

```

1  doctor << {
2    name = "Marco"
3    surname = "Rossi"
4    birthDate << {
5      day = 25
6      month = 5
7      year = 1971
8    }
9    specialization[0] = "Cardiology"
10   specialization[1] = "General Surgery"
11   email = "marco.rossi@gmail.com"
12 }
```

Fault handling

Jolie has constructs to handle faults. The primitives meant for this purpose are: `scope`, `install` and `throw`. For more details see [1].

2.1.2 Deployment

The deployment is the part of a Jolie program in which all the necessary information for establishing communication channels between services are established. The basic primitives are *input ports* and *output ports* which are based on *data types* and *interfaces*. A deployment D is simply a list of deployment instructions among which we can have input and output ports, type definitions, and interfaces [1]:

$$\begin{aligned}
 D ::= & IP \ (inputPort) \\
 & | OP \ (outputPort) \\
 & | T_{def} \ (typeDefinition) \\
 & | I \ (interface) \\
 & | \text{execution}\{ M \} \\
 & | E
 \end{aligned}$$

Communication ports

Communication ports are primitives that define *how* communications with other services are handled. There are two kind of ports. (*inputPort*) that exposes input operations to other services. Alternatively, (*outputPort*) permits the invocation of other services' operations.

The syntax for input and output port is the following.

$$\begin{aligned}
 IP & ::= \textcolor{red}{inputPort} : id \{ \\
 & \quad \textcolor{red}{location} : URI \\
 & \quad \textcolor{red}{protocol} : p \\
 & \quad \textcolor{red}{interfaces} : I_1, I_2, \dots, I_n \\
 & \quad [\textcolor{red}{aggregates} : OPid_1, \dots, OPid_n] \\
 & \}
 \end{aligned}$$

$$\begin{aligned}
 OP & ::= \textcolor{red}{outputPort} : id \{ \\
 & \quad \textcolor{red}{location} : URI \\
 & \quad \textcolor{red}{protocol} : p \\
 & \quad \textcolor{red}{interfaces} : I_1, I_2, \dots, I_n \\
 & \}
 \end{aligned}$$

These ports are based on three concepts of *location*, *protocol* and *interface*. The former two define the concrete binding between the program and other services. In particular the *location* must specify the communication medium. The mediums supported are: *local* (jolie in-memory communication), *socket*, *btl2cap* (Bluetooth L2CAP), *rmi* (Java RMI) and *localsocket* (Unix local sockets). The *protocol* indicates the data format. Jolie supports the following protocols: *HTTP*, *HTTPS*, *JSON-RPC*, *XML-RPC*, *SOAP*, *SODEP* (created and developed for Jolie), *SODEPS* and *RMI*. More details about the mediums and protocols can be found at [2]. The *interface* defines a bind between the deployment and the behavior: more specifically it defines type information that is expected to be satisfied by the behavior that uses the ports.

Data types and interfaces

Interfaces are a collection of operation types. The latter defines the type of the data to be communicated over each specified operation. Following the data type syntax.

$$T_{def} ::= \textcolor{red}{type} \ id \ T$$

$$T ::= : \ BT \ [\{ id_1 \ R_1 \ T_1 \ \dots \ id_n \ R_n \ T_n \}] \ | \ \textcolor{red}{undefined}$$

$$R ::= [min, max] \ | \ *$$

$$BT ::= \text{int} \mid \text{string} \mid \text{void} \mid \text{bool} \mid \text{long} \mid \text{double} \mid \text{raw} \mid \text{double}$$

A data type describes the structure of a data tree, the type of its nodes and the cardinality of each node, namely the number of admitted occurrences of a node. Basing on the `doctor` data tree, I give its type definition.

```

1  type Doctor: void {
2    name: string
3    surname: string
4    age : Date
5    specialization*: string
6    email[0,1]: string
7  }
8
9  type Date: void {
10   day: int
11   month: int
12   year: int
13 }
```

Doctor's root has type `void`, thus it must not contain any value. Whereas its nodes, except for `email`, must contain values of the specified type. `email` is an optional node, as its cardinality suggests, hence it is not mandatory to insert it in the data tree. The cardinality, if omitted, is set as `[1,1]` (which can be read "from one to one occurrences"). The `*` defines an unlimited number of occurrences, more in detail, it is a syntax sugar for `[1,*]` ("from one to unlimited occurrences"). The cardinality `[0,1]` can be written also with the shortcut `?`. Note that since Jolie 1.8.0 release, if the type of the root is `void`, there is no necessity to express it. Therefore the above defined type can be written as it follows.

```

1  type Doctor {
2    name: string
3    surname: string
4    age : Date
5    specialization*: string
6    email?: string
7  }
8
9  type Date {
10   day: int
11   month: int
12   year: int
13 }
```

More generally, a type T , can be either a basic type with (optionally) a list of named sub-nodes BT , or `undefined`, which makes the type accepting any sub-tree. Each sub-type comes with its *id*, its cardinality (or range) R , which can be an interval from *min* (integer greater or equal than zero) to *max* (integer greater or equal than *min* or `*`) and, finally, its type T .

The interface syntax is defined as it follows.

$$I ::= \text{interface } id \{ [\text{OneWay} : OW_1, \dots, OW_n] [\text{RequestResponse} : [RR_1, \dots, RR_n]] \}$$

$$OW ::= id(OT)$$

$$RR ::= id(OT_{req})(OT_{res})$$

$$OT ::= BT | \text{customTypeName}$$

An interface I is a list of possible one-way (OW) and request-response (RR) operation declarations. An OW definition consists of an operation name id and the type OT of its message. Similarly, an RR definition is composed by its name and both the type of the request message OT_{req} and the type of the response message OT_{res} . The deployment permits the type checking of the behaviors. Whenever a message is sent or received through a port, its type is checked against the one specified for its operation in the port's interface. If the type of a message received through an input port does not match to the one defined in the interface, a **TypeMismatch** fault is sent to the invoker. The same fault is thrown for an output statement trying to send a message with the wrong type.

Behavior instances

A service participates in a session by executing an instance of its behavior [1]. A session can be executed *one time*, multiple times in *sequence* or multiple times in *parallel*. For this purpose, Jolie allows to reuse behavioral definition multiple times with the *execution modality* deployment primitive before shown **execution**{ M } where M syntax is defined as it follows:

$$M ::= \text{single} | \text{sequential} | \text{concurrent}$$

single allows the execution of the behavior one time. **sequential** permits the execution of the behavior multiple times, more precisely it causes the program to be available again after the current instance is terminated. Finally, **concurrent** causes the program behavior to be instantiated and executed whenever its first input statement can receive a message [1]. In the latter two cases, the behavioral definition must be an input statement or an input choice. The state of a single behavior instance is not shared between the other instances, meaning that all the variables used are local to the behavior instance. Despite this, Jolie provides global variable support through the keyword **global**, which is used as a prefix on variable names. Following an example:

```
1  global.x = 5 //global variable x
2  y = 3 // local variable y
```

This permits sharing data between different behavior instances, which access can be restricted through **synchronized** blocks

$$B ::= \dots | \text{synchronized}(token) \{ B \}$$

that allows only one process at a time to enter any **synchronized** with the same *token*.

Architectural composition

Architectural composition is a different kind of composition that a deployment definition can obtain abstracting from the specific behavioral definitions of the involved services [1]. Architectural composition can be divided in two categories:

- **THE EXECUTION CONTEXT:** a service may execute other services in the same execution context. The linguistic primitive which allows the programming of execution contexts is the embedding [2] which can be distinguished between *static* and *dynamic*. In this thesis only the static embedding is presented, for more details and the dynamic embedding see [1, 2];
- **THE COMMUNICATION TOPOLOGY:** it permits the programming of the connections between services in a micro-service architecture [2]. The primitive presented is the *aggregation*, while *redirection*, *couriers* and *collection* can be studied at [1, 2].

Static embedding is a mechanism for executing multiple services in the same virtual machine. A service, called *embedder*, can embed another service, called *embedded* service, by targeting it with the *embedded* primitive [1].

$$E ::= \text{embedded} \{ E_{type} : \text{path} [\text{in } OP] \}$$

$$E_{type} ::= \text{Jolie} \mid \text{Java} \mid \text{JavaScript}$$

E_{type} specifies the technology of the service to embed, $path$ is a URL pointing to the service definition to embed. Jolie currently supports the three technologies mentioned in the syntax definition, making embedding a *cross-technology* mechanism. An output port OP may be optionally specified. In this case, as soon as the service is loaded, the output port OP is bound to the "local" communication input port of the embedded service [1]. For the specific case of embedding a Jolie service into another, the communication medium local must be explicitly indicated in the input port of the embedded service, with no necessity of specifying the protocol, as local medium use an in-memory communication. In an hierarchical perspective, the *embedder* is the parent service of the embedded ones. As a consequence, whenever a service is terminated, all his embedded services are recursively terminated. The hierarchy is also useful for performance: in-memory communications are faster as the services are running inside the same virtual machine. Following a simple example of how embedding with Jolie works.

```

1  //service that computes the length of a string
2  interface lengthInterface {
3      RequestResponse: length( string )( int )
4  }
5  inputPort LengthService {
6      location: "local"
7      interface: lengthInterface
8  }
9  main
10 {
11     length( request )( response ) {
12         response = #request

```

```

13     }
14 }

```

```

1  //embedder service
2  include "console.iol"
3
4  interface lengthInterface {
5      RequestResponse: length( string )( int )
6  }
7
8  outputPort LengthService {
9      interfaces: lengthInterface
10 }
11
12 embedded {
13     Jolie: "length_service.ol" in LengthService
14 }
15
16 include "console.iol"
17 main {
18     string = "Hello World!"
19     length@LengthService( string )( result )
20     println@Console( result )()
21 }

```

Aggregation is a generalization of network proxies that allows a service to expose operations without implementing them in its behavior, but instead delegating them to other services [1]. The syntax, as shown previously, *aggregates* : $OPid_1, \dots, OPid_n$ extends that for input ports, where *OPid* is the *id* of an output port. The interfaces of the aggregated output ports must not share any operation name. Whenever an input port *IP* receives a message for operation *opName*, there are three possible cases:

- *opName* is an operation declared in one of the interfaces of *IP*. In this case, the message is treated normally as previously described.
- *opName* is declared in the interface of an output port *OP* aggregated by *IP*, hence the message is forwarded to *OP* as an output message of the *aggregator*.
- Neither in the interface of the aggregated output ports nor in the *IP*'s interfaces *opName* is declared. Consequently, the message is rejected and an `IOException` fault is sent to the invoker.

Aggregation, therefore, is a mechanism that merges of the aggregated output ports and make them accessible through a single input port. From the invoker perspective, all the aggregated services are seen as a single one. Following an example of aggregation and embedding.

```

1  outputPort A {
2      location: "socket://someurlA.com:80/"

```

```
3    protocol: soap
4    interfaces: InterfaceA
5  }
6
7  outputPort B {
8    location: "socket://someurlB.com:80/"
9    protocol: jsonrpc
10   interfaces: InterfaceB
11  }
12
13  embedded { Java: "example.serviceB" in B }
14
15  inputPort M {
16    location: "socket://urlM.com:8000/"
17    protocol: sodep
18    aggregates: A, B
19  }
```

Observe that service B is actually a Java service (for details regarding Java services, see reference [2]), therefore it must be embedded in a output port in order to be able to forward messages to it.

2.2 Language Server Protocol

Implementing advanced features like auto-complete, hover information, diagnostics and go to definition for a programming language can take a significant effort. Typically, this work has to be repeated for every editor or IDE as each of them might offer different APIs for implementing the same feature. A Language Server is a software program that is intended to provide the aforementioned language-specific features and communicate with a development tool through a communication protocol.

The Language Server Protocol was developed by Microsoft to standardize the protocol for how such servers and IDEs communicate. Therefore, on one hand, the single Language Server can be reused for each development tool that implements the aforementioned protocol, on the other hand, the editor can support multiple languages with minimal effort.

2.2.1 How LSP works

A Language Server runs as a separate process and the development tool communicates with the server using JSON-RPC, a remote procedure protocol which uses JSON syntax as data-format. An LSP message is composed by two parts: the *header* part and the *content* part. The first part is composed by

Table 2.1: LSP headers

Header Field Name	Value Type	Description
Content-Length	number	The length of the content part in bytes. This header is required.
Content-Type	string	The mime type of the content part. Defaults to application/vscode-jsonrpc; charset=utf-8

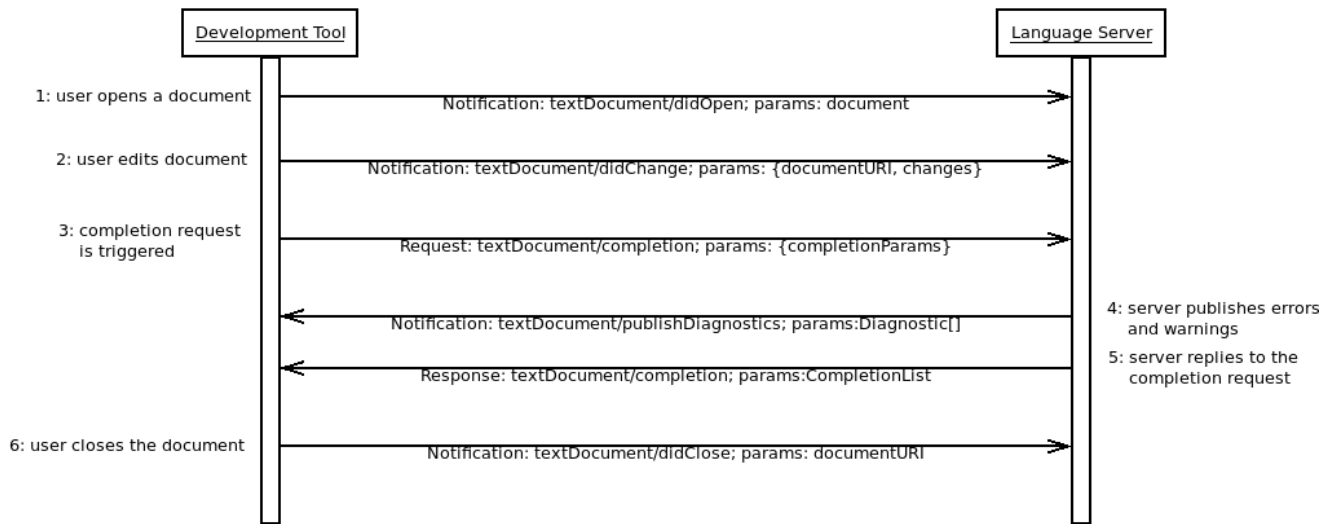
The header part is encoded using the 'ascii' encoding. This includes the '\r\n' separating the header and content part.

The content part contains the actual content of the message using JSON-RPC to describe requests, responses and notifications, the latter also called one-way messages. Following an example:

```

1 Content-Length: ... \r\n
2 \r\n
3 {
4   "jsonrpc": "2.0",
5   "id": 1,
6   "method": "textDocument/didClose",
7   "params": {
8     ...
9   }
10 }
```

The Language Server is started by the development tool when certain conditions occur (e.g. when it detects the language in question), therefore the server is required to be spawned as a child process of the client, and the communication between the two can be either via *stdio*, *ipc* or *socket*. More precisely, it is needed to add an extension that must include all the necessary configurations (for which language, conditions that must trigger the extension to start, etc.) in the development tool. This extension must be written in TypeScript/JavaScript and must override the method *activate* of the LSP libraries. This method states all the server and client options, namely the server module, the transport kind (and all the information required accordingly to specific kind of transport), the document selector and other options. The aforementioned extension is invoked when the triggering conditions aforementioned occur. Following a simplified sequence diagram that shows a slice of interaction between a client and a server.



1. **The user opens a file (referred as a *document*):** the tool notifies the server that a document is open ("textDocument/didOpen"), meaning that it is kept in the tool memory. The server will therefore save the language representation of the document in his memory.
2. **The user edits the document:** the tool notifies the server about a change made by the user on the document ("textDocument/didChange"). The server updates the language representation of the document.
3. **The tool execute the "Completion Request":** the request ("textDocument/completion") is triggered by the aforementioned changes. The parameter, *completionParams*, is essentially a JSON object with all the necessary information (documentURI, position, triggerKind) that the server needs to compute the response. Note that this call is *asynchronous*: the tool can send other notifications or request while waiting for the completion response, same for the Language Server.
4. **The server publishes errors and warnings:** after the change notification, the server computes and notifies the tool with a list of eventual errors and warnings in the code ("textDocument/publishDiagnostics"). The parameters of this notification contains all the necessary information in order to permit the tool flags errors and warnings.
5. **The server replies to the "Completion Request":** the server after computing a list of possible completion items, replies to the completion request made before by the tool. The latter lists the items received in order to let the user decide which suites the best.
6. **The user closes the document:** the tool notifies the server when the user closes the document, which cease to exist in the tool memory. The server erases all the information regarding the aforementioned document.

All the data types illustrated are language agnostic, meaning that can be applied to all the programming languages. This is due to the protocol simplicity: it is more simple to standardize a document URI or a position inside a it, that standardizing an abstract syntax tree and compiler symbol across different programming languages.

The aforementioned simplicity is shown in the following JSON-RPC objects that refers to the completion request and response:

Request:

```

1 {
2   "jsonrpc": "2.0",
3   "id" : 1,
4   "method": "textDocument/completion",
5   "params": {
6     "textDocument": {
7       "uri": "file:///home/user/Desktop/client.ol"
8     },
9     "position": {
10      "line": 3,
11      "character": 7
12    },
13    "context": {
14      "triggerKind": 1
15    }
16  }
17 }
```

Response:

```

1 {
2   "jsonrpc": "2.0",
3   "id": 1,
4   "result": {
5     "isIncomplete": false,
6     "items": {
7       "label": "println@Console",
8       "kind": 2,
9       "insertText": "println@Console( 1)(2 )",
10      "insertTextFormat": 2
11    }
12  }
13 }
```

A Language Server in order to work with LSP does not need to implement all the features. LSP provide **capabilities**. A capability groups a set of language features. The protocol permits the server to notify the tool which capabilities supports, therefore the tool adapts itself in order to send notifications and requests of the capabilities supported.

Analysis of the problem

In this chapter I will identify the requirements, gathered during the analysis of the Language Server Protocol and over a series of meetings with the supervisor.

3.1 Language Server Protocol features

LSP supports six key feature which are:

1. **Code completion:** feature that speeds up the process of coding applications by reducing typos and other common mistakes;
2. **Hover information:** feature that shows information, like the type signature, when the user moves the pointer over an element (such as a function definition);
3. **Jump to definition:** feature that shows the definition of a selected symbol;
4. **Workspace symbols:** displays all the symbols of the workspace, in order to help the user search for elements inside the workspace (such as classes, variables, methods, etc.);
5. **Find references:** given a symbol, this features lists all the project wide references;
6. **Diagnostics:** the tool flags syntax errors, warnings together with a description.

The protocol supports many other features that are all linked to the above listed. The requirements elicitation phase was based on these main ones.

3.2 Functional Requirements

Functional requirements are statements of services the system should provide, particularly how it should react to particular inputs [4]. From the features analyzed in the previous section, we extracted the following functional requirements, listed in ascending order of importance.

Table 3.1: Functional Requirements

Functional r. No.	Description
FR 1	The server must provide information of eventual programming errors and warnings every time a document is opened/modified.
FR 2	Every time the user starts typing an operation name, the server returns a list of possible completion items that consists of the full operation name and the output port.
FR 3	Every time the user starts typing a reserved word, the server provides a list of possible completion items.
FR 4	Every time a document is opened/modified/closed, the server saves/updates/deletes the information in his memory like the text, URI and a data structure containing all the information regarding the Jolie program (an abstract syntax tree of the <i>behavior</i> and data regarding the <i>deployment</i>).
FR 5	The server provides the type signature to the client every time the latter sends an hover request.
FR 6	The server computes and sends the operation definition every time it receives a definition. request
FR 7	The server provides a list or hierarchy of symbols of a specific document requested by the client.
FR 8	The server resolves project wide references of a given symbol.

3.3 Non-functional requirements

Non-functional requirements are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems [4].

Regarding the aforementioned non-functional requirements, the response time is critical for the project, as the server should be able to respond to the client as soon as possible. Nonetheless, this kind of calculations (such as code completion) are not computationally heavy.

Other non-functional requirements, are listed below.

3.3.1 Complying with the Language Server Protocol specification

The Language Server Protocol defines a common language, between a tool and a Language Server. As a consequence, the latter must be able to recognize different requests received from the client and send back proper responses that the client is able to interpret and compute.

Following an example: the request-response method `initialize`, that provides the server client's capabilities and it expects a response with the server capabilities, has the following type:

Request type

```

1 interface InitializeParams {
2   processId: number | null;
3   rootPath?: string | null;
4   rootUri: DocumentUri | null;
5   initializationOptions?: any;
6   capabilities: ClientCapabilities;
7   trace?: 'off' | 'messages' | 'verbose';
8   workspaceFolders?: WorkspaceFolder[] | null;
9 }

```

Response type

```

1 interface InitializeResult {
2   /**
3    * The capabilities the language server provides.
4    */
5   capabilities: ServerCapabilities;
6 }

```

JSON-RPC 2.0 Protocol

Regarding the communication protocol, LSP supports a custom version of the JSON-RPC 2.0 protocol, as mentioned in Chapter 2. A message is characterized by an header part, which can contain up to two headers (Content-Length, which is mandatory, and Content-Type) and a content part, with the actual message represented with the JSON-RPC notation.

Jolie supports JSON-RPC, but it runs over HTTP, therefore Jolie's JSON-RPC will expect HTTP headers before the Content-Length and Content-Type. As a consequence, Jolie's JSON-RPC protocol must be extended in order to make it compatible with LSP messages.

3.3.2 Distributed

The server must be adapted to the LSP, in order to make it works with different tools that uses the protocol concerned. Therefore, it must be **editor-agnostic** and it should be designed and implemented in order to be deployed both locally and in a different machine with respect to the clients, so it can interact with different tools at the same time. The LSP specifications indicates that when the development tool starts the plug-in, the latter will immediately start. Consequently the best communication channel to use is the socket.

3.3.3 Modular

Not every Language Server can support all features defined by the protocol. LSP therefore provides, the previous mentioned, **capabilities**, as stated in Chapter 2. A capability groups a set of language features. A development tool and the Language Server announce their supported features using capabilities. For instance, the server announces that it can handle the `textDocument/hover` request, but it might not support the `textDocument/references` request. Similarly, a development tool announces its ability

to provide `textDocument/didChange` notification when a document is modified, so that a server can compute textual edits to format the edited document.

On account of this, the server must be designed in order to easily add new capabilities or improve the existing ones. After implementing a new feature, the programmer just need to modify the server capabilities thus the client can start sending requests regarding the newly activated capability.

Table 3.2: Non-Functional Requirements

Non-functional r. No.	Description
NFR 1	The server must support the LSP's JSON-RPC protocol.
NFR 2	The server must be designed to be modular.
NFR 3	The server must be able to respond to a client request as soon as possible.
NFR 4	The server must work both when deployed in a different machine and when deployed in a local machine, with respect of the client location. As a consequence, it has to support the socket channel of communication.
NFR 5	The server if distributed, must be able to handle more clients sending multiple requests at the same time.

System architecture

The architecture of a system identifies the components that form a system and the relationships between them. Graphical models are the best technique to abstractly represent a system architecture. Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch, 2000) [4]. In order to satisfy the non-functional, in particular NFR 2 and NFR 5 requirements defined in Chapter 3, the architecture adopted is the Service Oriented Architecture. Instead of a monolithic architecture, SOC is more scalable: the direct communication with the client is handled by a single service which deals with the initialization and the termination of the entire server while it forwards all the other requests and notifications sent by the development tool to the service responsible for treating the specific message (e.g. the message `textDocument/didOpen`, when received by the main service, named *orchestrator*, forwards the message to the `textDocument` service, namely the service that handles all the `textDocument` messages).

4.1 Architecture

In this section I present two architectures according to NFR 4. One architecture, represents the system with the Language Server deployed in the same machine as the development tool, while the other describes the architecture of the system when the Language Server is deployed in a different machine than the client.

Figure 4.1: Architecture in L.S. and dev. tool in a single machine

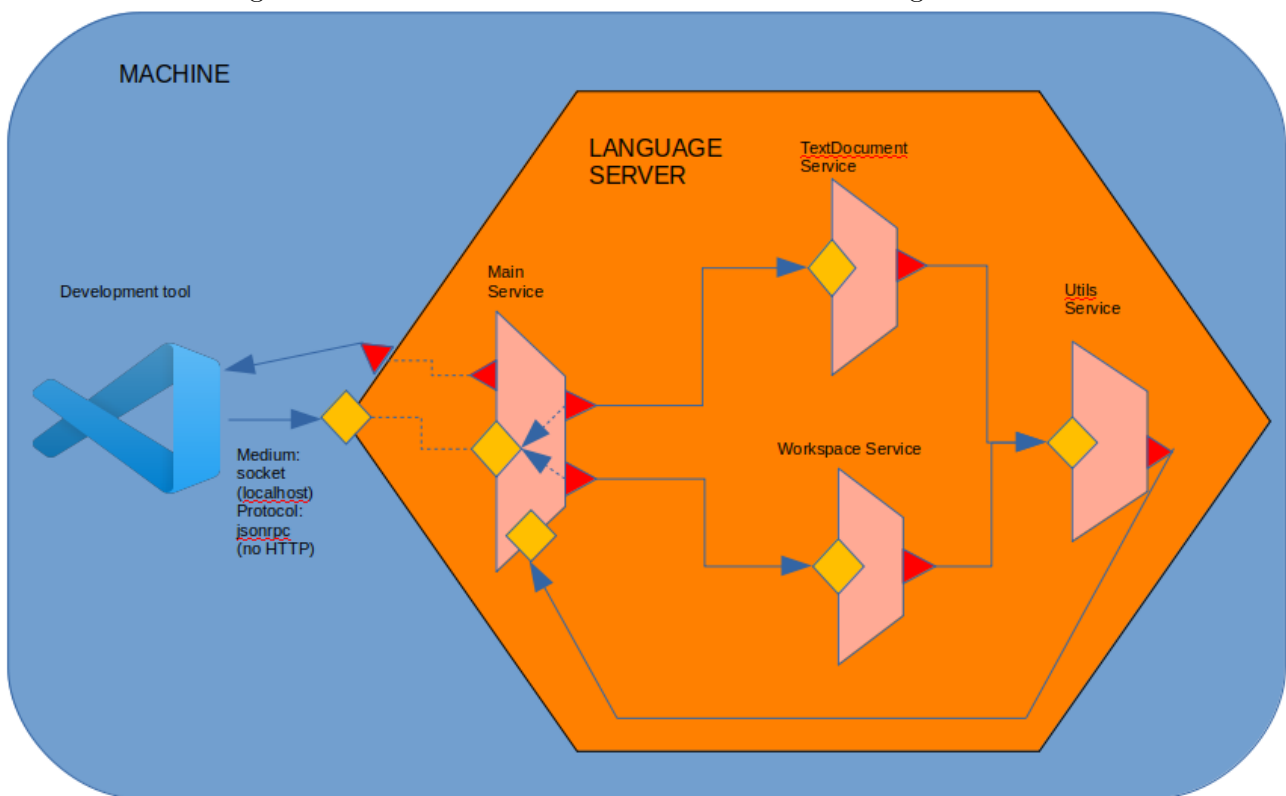


Figure 4.2: L.S. and dev. tool in different machines

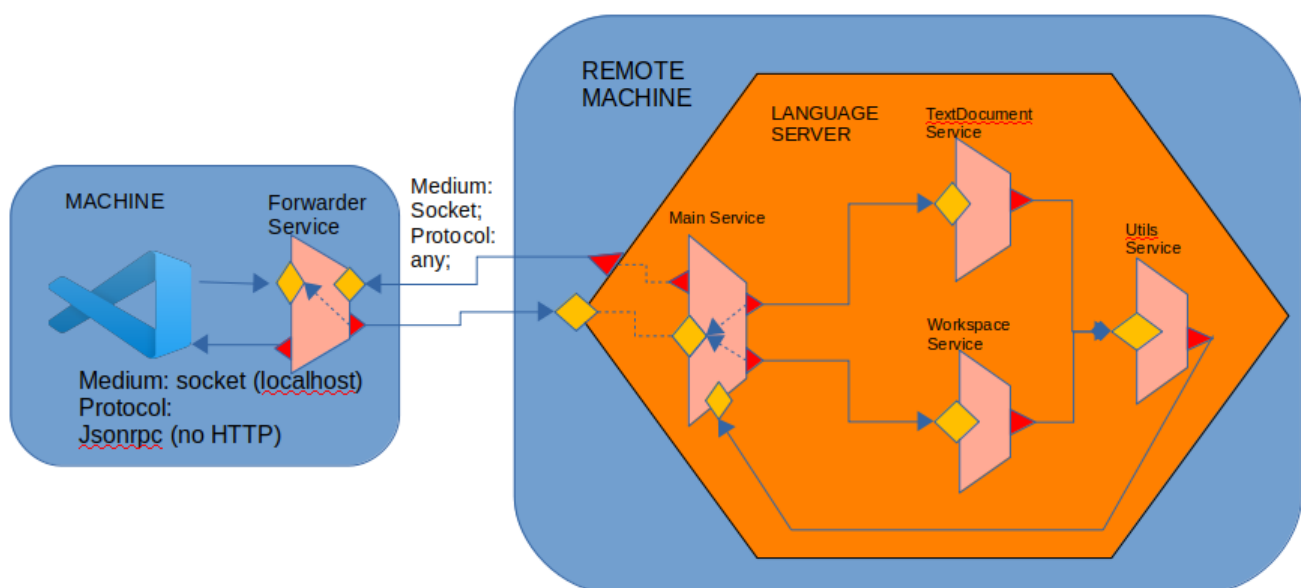
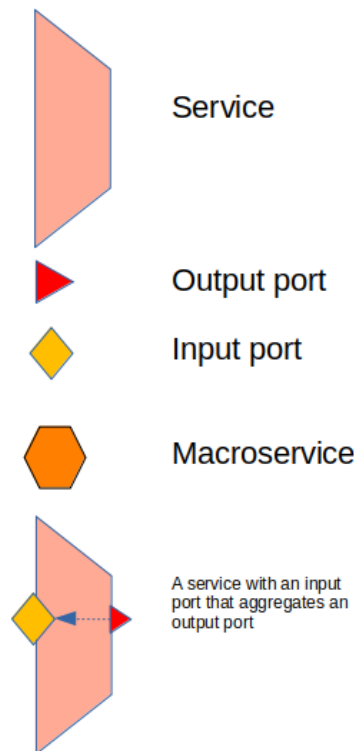


Figure 4.3: Legenda



The hexagon represents a *macroservice*, which is an unique execution context for a set of services. A macroservice exhibit only the public available ports of the inner services [2].

4.2 Services

Each service presented in the model has a set of operations. The main difference between the two models is that in the distributed model there is one more service deployed in the development tool machine that acts as a forwarder. This is necessary as the client process expects a child process that contains the server. The client then will communicate with this service that forwards all its messages to the real Language Server deployed in a remote machine.

4.2.1 Main service

The main service presents two input ports and three output ports. One input port is used to receive the requests and notifications sent by the client, the other is used by an another service, called *utils*, with the purpose to receive all the notifications to be sent to the client through one of the output ports. The remaining two output ports, point to the two main services of the server, *textDocument* and *workspace* and they should be aggregated by the first mentioned input port, therefore every message si automatically forward to the respective service.

The operations offered by this service are those necessary for initialization and termination of the server. For more details see reference [3]. The service must handle at least the following requests and notifications:

- **initialize** request which content contains the *client capabilities*. The response consists of the

server capabilities;

- **initialized** notification. An empty message acting as an ACK for the **initialize** response;
- **shutdown** request. Empty message that expects an empty response;
- **exit** notification. After receiving the **shutdown** response, a notification is sent and the server must terminate its execution.

4.2.2 Text Document service

The service *textDocument* handles all the text document messages. In order to meet the functional requirements the service must serve the following messages:

- **textDocument/completion** request. This request is sent with a cursor position and expects a response with a list of completion items that will be presented in the user interface;
- **textDocument/didOpen**, **textDocument/didChange**, **textDocument/didClose** notifications. Messages used for synchronize the document's truth with the language server. The service should forward the message to the service *utils* that will respectively store, update or delete the information regarding the document (URI, text, etc.);
- **textDocument/hover** request is sent together with a cursor position thus the service could compute a response with the information of what is located in that document position.
- **textDocument/documentSymbol** request requires a flat list or hierarchy of a symbols of a specified document.
- **textDocument/definition** request. The client requires the server to resolve the definition location of a given document symbol at a given text document position.
- **textDocument/references** request is sent in order to resolve project-wide reference of the symbol denoted by the given text document position.

For all the text document messages, see reference [3].

4.2.3 Workspace service

All the workspace messages, are handled by this service. More details regarding this type of messages can be studied at reference [3].

4.2.4 Utils service

Utils service provide functionalities such as syntax checking and document synchronization. For each time a document synchronization message, e.g. **textDocument/didChange**, is received by *textDocument* service, the latter forwards it to *Utils* service that keeps a data structure with all the documents that are open in a specific moment. Subsequently, a syntax check is executed on the updated or newly opened document, and the result is sent to the main service which in turn forwards it to the client.

4.2.5 Forwarder service

The *forwarder* service is an additional service used for the distributed system. The input port exposed to the development tool aggregates the output port used for contacting the server. Jolie is *protocol agnostic*. This means that the communication between the forwarder and the Language Server does not need to use the JSON-RPC protocol: the aggregator, which in our case is the forwarder, automatically transforms the messages thus granting a transparent composition of the server.

Implementation

For implementing the Language Server for Jolie, modeled in the previous chapter, I used the language Jolie itself for the following reasons:

- **Service orientation:** Jolie, as described in Chapter 2, is microservice-oriented language, which is perfect for the non-functional requirements described in Chapter 3, in particular NFR 1, as Jolie already supports the JSON-RPC protocol over HTTP, therefore it must be adapted to the protocol used by LSP, by extending its functionalities. Moreover, Jolie has a service included in the standard library, called *Inspector*, that provides code inspection operations for analyzing Jolie source code, therefore useful in order to meet FR 1 and FR 2.
- The macroservice concept is realized with the *static embedding* primitive: the main service embeds all the other three services, therefore all four services will run in the same virtual machine. This will reflect in faster internal communications. Furthermore, the main service *aggregates* the output ports pointed respectively to the services *workspace* and *textDocument* to guarantee an automatic forward of the messages that are not of its concern.

The implementation phase was based on the local architecture model described in the previous chapter and it was characterized by the following two steps:

1. **Complying with the LSP's JSON-RPC specifications:** LSP uses a simplified version of JSON-RPC protocol, whereas Jolie internal JSON-RPC protocol uses it over HTTP. Therefore, the protocol was extended in order to be capable of handling also LSP messages. Moreover, the Jolie interpreter, when receiving a JSON-RPC message is able to redirect it to the correct operation by using the *method* field in the JSON-RPC message. Despite this, most of the method names in LSP have prefixes, for instance `textDocument/didOpen`, which is problematic as operation names in Jolie does not support special characters such as the `"/`. As a consequence, Jolie's protocol must be extended in order to alias the operation names properly.
2. **Implementation of the Language Server:** define all the services with their deployment parts (ports, types and interfaces) and their behavior (computations).

5.1 Complying with LSP JSON-RPC specifications

5.1.1 LSP message and parser

In order to extend Jolie JSON-RPC protocol, *protocol parameters* were exploited. Input port protocol definitions could be extended with parameters, namely a list of options regarding compression, debug settings, etc.. The parameter added, called *transport* is a simple string. Following an example of an input port using JSON-RPC with the aforementioned parameter.

```

1  inputPort Input {
2    location:JolieLS
3    protocol: jsonrpc {
4      transport = "lsp"
5    }
6    interfaces:...
7    ...
8  }
```

Jolie, when it identifies this parameter, must not use JSON-RPC over HTTP, but according to LSP specification. In order to do this, the modifications made in the Jolie protocol are:

- Definition of two new classes: `LSPMessage` and `LSPParser`. The first represents an LSP message, according to the specifications defined in Chapter 2, the second is a parser used on the data read from the buffer, in order to extract headers and the JSON object properly, therefore, the latter is parsed by the `JsUtils` class in order to obtain a Jolie value representation of the original JSON message.
- Modified `JsonRpcProtocol` class definition which now checks if there is the parameter *transport* with value "lsp", and, according to the result of this check, uses the appropriate parser (HTTP or LSP).

Following the `LSPMessage` class definition:

```

1  public class LSPMessage {
2
3    private byte[] content = null;
4    final private Map< String, String > propMap = new HashMap<>();
5
6    public String getProperty( String name ) {
7      String property = getPropMap().get( name.toLowerCase() );
8      if(property!=null)
9        property=property.trim();
10     return property;
11   }
12
13   protected Map<String, String > getPropMap() {
14     return propMap;
15   }
```

```

16     public void setContent( byte[] content ) {
17         this.content = content;
18     }
19
20     public void setProperty(String name, String value) {
21         propMap.put( name.toLowerCase(), value );
22     }
23
24     public Collection< Entry< String, String > > getProperties() {
25         return propMap.entrySet();
26     }
27
28     public int size() {
29         if ( content == null )
30             return 0;
31         return content.length;
32     }
33
34     public byte[] content() {
35         return content;
36     }
37 }

```

where the field `content` represent the JSON message, while `propMap` is an `HashMap` containing the header names and their relative values.

5.1.2 Aliasing LSP method names to Jolie operation names

Similarly to the problem presented in the previous section, aliasing must be handled at protocol level. Therefore we used the parameter `osc`, acronym for *operation specific parameter*, for each operation name to be aliased.

```

1  inputPort Input {
2      location:JolieLS
3      protocol: jsonrpc {
4          transport = "lsp"
5          osc.didOpen.alias = "textDocument/didOpen"
6          osc.didClose.alias = "textDocument/didClose"
7          osc.completion.alias = "textDocument/completion"
8          ...
9      }
10     interfaces:...
11     ...
12 }

```

The above example demonstrate that protocol parameter is a Jolie *data tree*. The parameter `osc` must have a child node with the name of the operation and in turn a child node name `alias` with a string

value that represents the alias.

The modifications made in the `JsonRpcProtocol` class were substantially the following:

- When **receiving** a message, after the parsing phase, given the LSP method name contained in the subnode `method`, compute the correspondent operation name in the Jolie program, by visiting the `osc` tree.
- When **sending** a message, the steps are analogue, but the aliasing is done before parsing the Jolie value to a JSON string and it is from the Jolie operation name to the LSP method name.

Following the code for aliasing operation names when receiving a message.

```

1  String operation = value.getFirstChild( "method" ).strValue();
2
3  // Resolving aliases
4  if ( hasParameter( Parameters.OSC ) ) {
5      Value osc = getParameterFirstValue( Parameters.OSC );
6      for( Entry<String, ValueVector> ev : osc.children().entrySet() ) {
7          Value v = ev.getValue().get( 0 );
8          if ( v.hasChildren( Parameters.ALIAS ) ) {
9              if ( v.getFirstChild( Parameters.ALIAS ).strValue().equals( operation ) ) {
10                 operation = ev.getKey();
11             }
12         }
13     }
14 }
```

Where the variable `operation` contains the name of the LSP method name (e.g. `textDocument/didOpen`), the variable `osc` is the root of the data tree. `Value` and `ValueVector` are the classes used to represent data trees in Jolie. Each single node of a tree in Jolie is a `ValueVector` containing one or more objects of type `Value` which contains basic type value and its children nodes, which in turn are `ValueVector` objects.

The code for aliasing an outgoing message is analogue.

```

1  String operationNameAliased = message.operationName();
2  //resolving aliases
3  if ( hasParameter( Parameters.OSC ) ) {
4      Value osc = getParameterFirstValue( Parameters.OSC );
5      for( Entry<String, ValueVector> ev : osc.children().entrySet() ) {
6          Value v = ev.getValue().get( 0 );
7          if ( v.hasChildren( Parameters.ALIAS ) ) {
8              if ( ev.getKey().equals( operationNameAliased ) ) {
9                  operationNameAliased = v.getFirstChild( Parameters.ALIAS ).strValue();
10             }
11         }
12     }
13 }
```

5.2 Server implementation

The server structure is the presented in Chapter 4, thus it will be composed by four Jolie programs together with the files with respectively the types definitions and the interfaces used by the various ports. The full implementation of the server can be viewed at the reference [5].

5.2.1 Operation types and interfaces

Though Jolie operations can be type as **undefined**, I preferred define all the operation types basing on the LSP specification, in order to type check every message to be received and sent. Therefore, **maintainability** will be affected positively, as changes in the type definitions of LSP messages that might be applied in future releases of the protocol, can be identified immediately as the type check could fail, sending a fault to the invoker or to the server itself.

Following the interface definitions for all the ports used in the service.

```

1  include "types/lsp.iol"
2
3  interface GeneralInterface {
4      OneWay:
5          initialized( InitializedParams ),
6          onExit( void ),
7          cancelRequest //cancelRequest( undefined )
8      RequestResponse:
9          initialize( InitializeParams )( InitializeResult ),
10         shutdown( void )( void )
11 }
12
13 interface TextDocumentInterface {
14     OneWay:
15         didOpen( DidOpenTextDocumentParams ),
16         didChange( DidChangeTextDocumentParams ),
17         willSave( WillSaveTextDocumentParams ),
18         didSave( DidSaveTextDocumentParams ),
19         didClose( DidCloseTextDocumentParams )
20     RequestResponse:
21         willSaveWaitUntil( WillSaveTextDocumentParams )( WillSaveWaitUntilResponse ),
22         completion( CompletionParams )( CompletionResult ),
23         hover( TextDocumentPositionParams )( HoverInformations ),
24         documentSymbol( DocumentSymbolParams )( DocumentSymbolResult ),
25         signatureHelp( TextDocumentPositionParams )( SignatureHelpResponse )
26 }
27
28 interface WorkspaceInterface {
29     OneWay:
30         didChangeWatchedFiles( DidChangeWatchedFilesParams ),
31         didChangeWorkspaceFolders( DidChangeWorkspaceFoldersParams ),
32         didChangeConfiguration( DidChangeConfigurationParams )

```

```

33     RequestResponse:
34         symbol( WorkspaceSymbolParams )( undefined ),
35         executeCommand( ExecuteCommandParams )( ExecuteCommandResult )
36     }
37
38     interface ServerToClient {
39         OneWay:
40             publishDiagnostics( PublishDiagnosticParams )
41     }
42
43     interface UtilsInterface {
44         RequestResponse:
45             getDocument( string )( TextDocument )
46         OneWay:
47             insertNewDocument( DidOpenTextDocumentParams ),
48             updateDocument( DocumentModifications ),
49             deleteDocument( DidCloseTextDocumentParams )
50     }

```

The full type definitions can be viewed at the reference [3].

5.2.2 Main service

The main service handles the initialization and termination of the server. Furthermore it *orchestrates* the messages sent by the development tool to the services that are concerned.

```

1  /*
2   * Main Service that communicates directly with the client and provides the basic
3   * operations
4   */
5   execution { sequential }
6
7   include "internal/deployment.iol"
8
9   include "console.iol"
10  include "string_utils.iol"
11  include "runtime.iol"
12
13  init {
14      Client.location -> global.clientLocation
15      println@Console( "Jolie Language Server started" )()
16      global.receivedShutdownReq = false
17  }
18
19  main {
20      [ initialize( initializeParams )( serverCapabilities ) {
21          println@Console( "Initialize message received" )()
22          global.processId = initializeParams.processId

```

```

23     global.rootUri = initializeParams.rootUri
24     global.clientCapabilities << initializeParams.capabilities
25     //for full serverCapabilities spec, see
26     // https://microsoft.github.io/language-server-protocol/specification
27     // and types.iol
28     serverCapabilities.capabilities << {
29         textDocumentSync = 1 //0 = none, 1 = full, 2 = incremental
30         completionProvider << {
31             resolveProvider = false
32             triggerCharacters[0] = "@"
33         }
34         //signatureHelpProvider.triggerCharacters[0] = "("
35         definitionProvider = false
36         hoverProvider = true
37         documentSymbolProvider = false
38         referenceProvider = false
39         //experimental;
40     }
41 } ]
42
43 [ initialized( initializedParams ) ] {
44     println@Console( "Initialization done " )()
45 }
46
47 [ shutdown( req )( res ) {
48     println@Console( "Shutdown request received..." )()
49     global.receivedShutdownReq = true
50 } ]
51
52 [ onExit( notification ) ] {
53     if( !global.receivedShutdownReq ) {
54         println@Console( "Did not receive the shutdown request, exiting anyway..." )()
55     }
56     println@Console( "Exiting Jolie Language server..." )()
57     exit
58 }
59 //received from utils.ol
60 [ publishDiagnostics( diagnosticParams ) ] {
61     println@Console( "publishing diagnostics for " + diagnosticParams.uri )()
62     publishDiagnostics@Client( diagnosticParams )
63 }
64
65 [ cancelRequest( cancelReq ) ] {
66     println@Console( "cancelRequest received ID: " + cancelReq.id )()
67     //TODO
68 }

```

69 }

The *deployment* part of this service is imported from another file with the primitive **include**. The *behavior* is an input choice, in which initialization and termination of the server are handled. When the service receives an **initialize** message, it executes the behavior of the input *initialize* where the client capabilities are saved in a data structure and the **serverCapabilities** are computed and sent to the invoker. Consequently, the client, after receiving the language server capabilities it sends an **initilized** notification with no content and the interaction can start. **Termination** follows the same steps: first, the client send an empty **shutdown** request, and after receiving the response, it sends a *exit* notification, aliased to **onExit** as **exit** is a reserved word, where the server must terminate its execution.

5.2.3 Utils service

Utils service has four input operations:

- **insertNewDocument**: after receiving a **textDocument/didOpen** message from the development tool, the document received is inspected through the *Inspector* service, which extracts information like output and input ports, their interfaces with the operation definitions and the type definition of the latters. Nonetheless, if the there is a syntax error in the document received, the *Inspector* provides a string containing all the information related to the aforementioned error, including the position. This is used two build a **diagnosticParam** structure to be sent to the main service previously described, which in turn forwards it to the client. When the latter receives the diagnostics, it will publish them through its UI. If the document does not have any error, the document data, namely the *uri*, the *source code*, the *inspection result* and an array containing the lines of code, are saved normally in a global variable. This variable is an array containing all the documents that opened in the development tool.
- **updateDocument**: the message received through the notification **textDocument/didChange** is forwarded to this input operation, that similarly achieves what **insertDocument** does. Unlikely the latter, **updateDocument** publishes also the empty diagnostics, in order to clear the client's UI when a syntax error is corrected by the user.
- **deleteDocument**: input operation that receives a document URI and deletes the array element containing the correspondent URI.
- **getDocument**: request-response operation that sends to the invoker the full data structure with the corresponding URI received. This operation is used by Text Document service in order to perform tasks like **completion** and **hover**.

5.2.4 Text Document service

The purpose of this service is to handles all the text document messages sent by the development tool, for instance **textDocument/didOpen** or **textDocument/completion**. Two main features has been implemented: **auto-completion** for operation definitions and keywords, and **hover** information of both input and output operation definitions.

Completion

Given a text document position, the server ought to compute a list of possible completion item to integrate what is already present the aforementioned position. I implemented this functionality for auto-completing output statements. For instance, if in the given text document position the server finds "pri", it ought to compute a list of output operations that are found in the interfaces of the output ports of the service. In this case, the server will return the following list: ["print@Console(\$0)(\$1)", "println@Console(\$1)(\$2)"]. These completion items are in the form of **snippets**. A **snippet** is a small and reusable piece of code, in which tab stops and placeholders can be defined with the "\$1", "\$2" and "\${3:foo}". "\$0" identifies the end of the snippet. Placeholders with equal identifiers are linked, that is typing in one will update the others too [3].

Following the code of the completion input operation.

```

1  /*
2   * RR sent sent from the client when requesting a completion
3   * works for anything callable
4   * @Request: CompletionParams, see types/lsp.iol
5   * @Response: CompletionResult, see types/lsp.iol
6   */
7  [ completion( completionParams )( completionRes ) {
8      println@Console( "Completion Req Received" )()
9      completionRes.isIncomplete = false
10     txtDocUri -> completionParams.textDocument.uri
11     position -> completionParams.position
12
13     if ( is_defined( completionParams.context ) ) {
14         triggerChar -> completionParams.context.triggerCharacter
15     }
16
17     getDocument@Utils( txtDocUri )( document )
18     //character that triggered the completion (@)
19     //might be not defined
20
21     program -> document.jolieProgram
22     codeLine = document.lines[position.line]
23     trim@StringUtils( codeLine )( codeLineTrimmed )
24     portFound = false
25
26     for ( port in program.outputPorts ) {
27         for ( iFace in port.interfaces ) {
28             for ( op in iFace.operations ) {
29                 if ( !is_defined( triggerChar ) ) {
30                     //was not '@' to trigger the completion
31                     contains@StringUtils( op.name {
32                         substring = codeLineTrimmed
33                     } )( operationFound ) // TODO: fuzzy search

```

```

34         undef( temp )
35         if ( operationFound ) {
36             snippet = op.name + "@" + port.name
37             label = snippet
38             kind = CompletionItemKind_Method
39         }
40     } else {
41         //@ triggered the completion
42         operationFound = ( op.name == codeLineTrimmed )
43
44         label = port.name
45         snippet = label
46         kind = CompletionItemKind_Class
47     }
48
49     if ( operationFound ) {
50         //build the rest of the snippet to be sent
51         if ( is_defined( op.responseType ) ) {
52             //is a reqRes operation
53             reqVar = op.requestType.name
54
55             resVar = op.responseType.name
56             if ( resVar == NATIVE_TYPE_VOID ) {
57                 resVar = ""
58             }
59             snippet += "( {1:" + reqVar + "} )( {2:" + resVar + "} )"
60         } else {
61             //is a OneWay operation
62             notificationVar = op.requestType.name
63
64             snippet = "( {1:" + notificationVar + "} )"
65         }
66
67         //build the completionItem
68         portFound = true
69         completionItem << {
70             label = label
71             kind = kind
72             insertTextFormat = 2
73             insertText = snippet
74         }
75         completionRes.items[#completionRes.items] << completionItem
76     }
77 }
78 }
79 }

```

```
80
81 //loop for completing reservedWords completion
82 //for (keyword in global.keywordSnippets.snippet)
83 //doesn't work, used a classic for with counter
84 keyword -> global.keywordSnippets
85 for ( i=0, i<#keyword.snippet, i++ ) {
86     contains@StringUtils( keyword.snippet[i] {
87         substring = codeLineTrimmed
88     } )( keywordFound )
89
90     if ( keywordFound ) {
91         completionItem << {
92             label = keyword.snippet[i]
93             kind = CompletionItemKind_Keyword
94             insertTextFormat = 2
95             insertText = keyword.snippet[i].body
96         }
97         completionRes.items[#completionRes.items] << completionItem
98     }
99 }
100
101 if ( !foundPort && !keywordFound ) {
102     completionRes.items = void
103 }
104 println@Console( "Sending completion Item to the client" )()
105 } ]
```

6

Validation

6.1 Met requirements (screenshots)

6.2 Unmet requirements

**Conclusions: review, what needs to be
done**

I

Appendici

A

Altro capitolo

Summary

Bibliography

- [1] Fabrizio Montesi and Claudio Guidi and Gianluigi Zavattaro. Service-oriented programming with jolie. <https://www.fabriziomontesi.com/files/mgz14.pdf>.
- [2] Jolie Development Team. Jolie documentation. <https://jolielang.gitbook.io/docs/>.
- [3] Microsoft. Language server protocol specification. <https://microsoft.github.io/language-server-protocol/specification>.
- [4] Ian Sommerville. *Software Engineering*. Pearson, 2011.
- [5] Jolie Development team. Jolie language support. <https://marketplace.visualstudio.com/items?itemName=jolie.vscode-jolie&ssr=false>.