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

1

Introduction

2

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:

$$Program \quad ::= \quad D \quad main\,\{\,B\,\}$$

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:

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

where

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

and

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

Communications

Communications are implemented through *Rules (input)*, *(output)* and *(inputChoice)*. Input $\boldsymbol{\eta}$ 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 `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 $\eta_i \neq \eta_j, \forall i, j \in [1, n] \mid i \neq j$, 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 `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.

```
x[0] = 1
x[1] = 2
x[2] = 3
println@Console( #x )() // prints 3

for( i = 0, i < #x, i++) {
  println@Console( x[i] )()
}
//which is equivalent to
for( el in x ) {
  println@Console( el )()
}
```

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.

```
doctor.name = "Marco"
doctor.surname = "Rossi"
doctor.age = 45
doctor.specialization[0] = "Cardiology"
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:

$$undef\,(\,variable\_name\,)$$

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 $<<$ 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.

```
doctor << {
  name = "Marco"
  surname = "Rossi"
  birthDate << {
    day = 25
    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]:

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

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.

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

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

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.

$$T_{def} \quad ::= \quad type \; id \; T$$

$$T \quad ::= \; : \; BT \left[\{ \; id_1 \; R_1 \; T_1 \; ... \; id_n \; R_n \; T_n \; \} \right] \mid undefined$$

$$R \quad ::= \quad [min, \, max] \mid *$$

$$BT \quad ::= \quad int \mid string \mid void \mid bool \mid long \mid double \mid raw \mid double$$

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.

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

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

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

$$OT \ ::= \ BT \,|\, customTypeName$$

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:

$$M \ ::= \ single \mid sequential \mid concurrent$$

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

$$B \ ::= \ ... \mid synchronized \, (\, token \,) \, \{ \, B \, \}$$

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].

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

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

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* : $OPid_1, ..., OPid_n$ 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 in 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 communicate 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

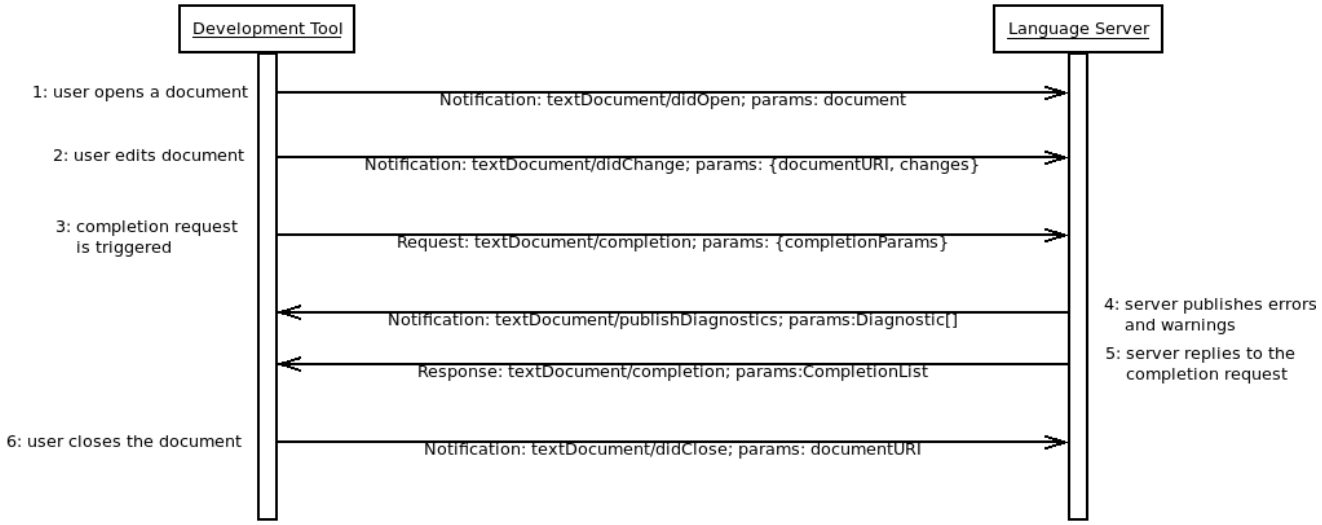
Table 2.1: LSP headers

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:

```
Content-Length: ...\r\n
\r\n
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/didClose",
  "params": {
    ...
  }
}
```

Following a simplified sequence diagram that shows a slice of interaction between a client and a server.



1. **The user opens a file (refferred 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:

```
{
  "jsonrpc": "2.0",
  "id" : 1,
  "method": "textDocument/completion",
  "params": {
    "textDocument": {
      "uri": "file:///home/user/Desktop/client.ol"
    },
    "position": {
      "line": 3,
      "character": 7
    },
    "context": {
      "triggerKind": 1
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
  "isIncomplete": false,
    "items": {
      "label": "println@Console",
      "kind": 2,
      "insertText": "println@Console( )( )",
      "insertTextFormat": 2
    }
  }
}
```

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.

3

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 [3]. 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 *behavior* and data regarding the *deployment*).
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	Information regarding the workspace are sent to the client and updated every time the client requests them (workspace symbols for example).
FR 9	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 [3].
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 capa-

bilities and it expects a response with the server capabilities, has the following type:

Request type

```
interface InitializeParams {
  processId: number | null;
  rootPath?: string | null;
  rootUri: DocumentUri | null;
  initializationOptions?: any;
  capabilities: ClientCapabilities;
  trace?: 'off' | 'messages' | 'verbose';
  workspaceFolders?: WorkspaceFolder[] | null;
}
```

Response type

```
interface InitializeResult {
  /**
   * The capabilities the language server provides.
   */
  capabilities: ServerCapabilities;
}
```

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. 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.

4

Design of the system

4.1 Architecture

4.1.1 Modules, interfaces, communication

4.2 What needs to be implemented

5

Implementation

5.1 Choosing a language: Jolie (why)

5.2 Implementation details

5.2.1 Complying with LSP's JSON-RPC protocol

The JSON-RPC 2.0 is a remote procedure call protocol that uses the JSON, a lightweight data-interchange format which is language independent. As stated in Chapter 2, an LSP message supports only two headers (*Content-Lenghth* and *Content-Type*), while the Jolie protocol runs over HTTP. This incompatibility led to a complex rework on the aforementioned Jolie protocol in order to meet the FR 1.
Further details can be read in Chapter 5.

6

Validation

6.1 Met requirements (screenshots)

6.2 Unmet requirements

7

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] Ian Sommerville. *Software Engineering*. Pearson, 2011.