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.

2.1.1 Language basics: behavior and deployment

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 Behaviour. The main procedure is the execution entry point.

Behaviour

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

$$
\begin{aligned}
B \quad ::= & \ \eta \quad (input) \\
& \mid \bar{\eta} \quad (output) \\
& \mid if(e)\ B_1\ [else\ B_2] \quad (condition) \\
& \mid while(e)\ B \quad (while) \\
& \mid for(init,\ cond,\ incr)\ B \quad (for) \\
& \mid for(x\ in\ y)\ B \quad (array\ iteration) \\
& \mid foreach(c : t) \quad (foreachchildintree) \\
& \mid B_1\ ;\ B_2 \quad (sequential) \\
& \mid B_1 \mid B_2 \quad (parallel) \\
& \mid \{\ B\ \} \quad (block) \\
& \mid x\ =\ e \quad (assign) \\
& \mid [\ \eta_1\ ]\ \{\ B_1\ \} ... [\ \eta_n\ ]\ \{\ B_n\ \} \quad (inputChoice)
\end{aligned}
$$

where

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

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

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.

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

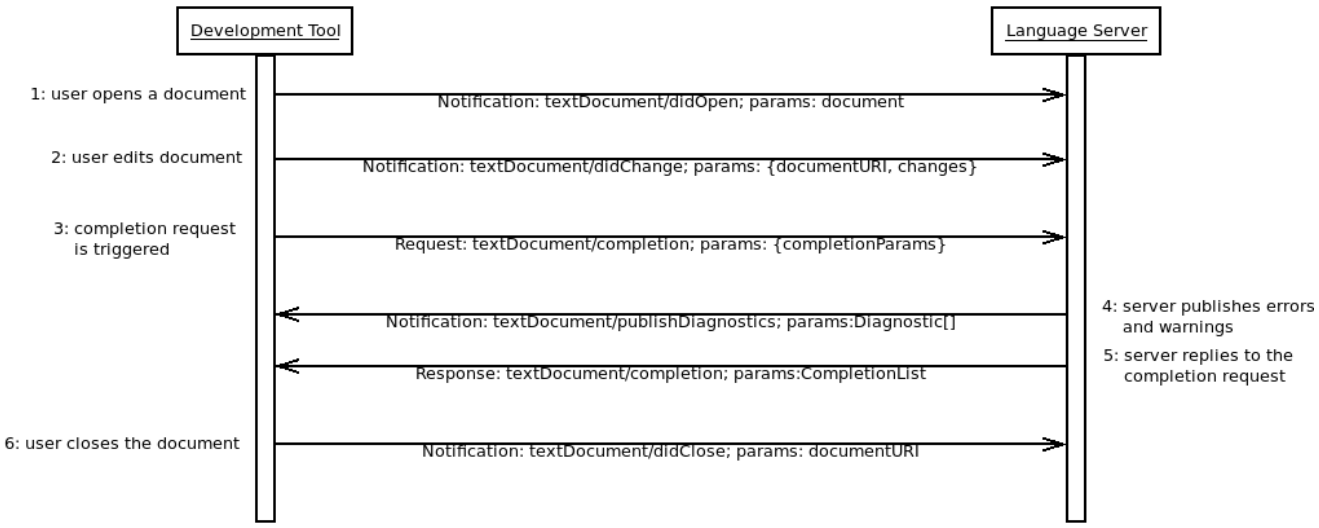
Table 2.1: LSP headers

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 [2]. 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 [2].
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, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. page 1.

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