Software Engineering 2

PZotIDE

Version 1.1

Prof. Matteo Rossi

Alessandro Pozone - 10399281

Academic Year 2017 - 2018

POLITECNICO
MILANO 1863

Contents

Chapter 1

Introduction

1.1 Introduction

PZotIDE is a tool made to efficiently write PZot formulas.
It is composed by a text editor and a dependency graph editor that allow to edit formulas or dependencies both by code and in a graphical way.

For more details about the PZot language we suggest to read the PZot manual.[1]

1.2 Revision History

Version	Release Date	Description
1.0	27/02/2018	Initial Release
1.1	23/10/2018	Updated the Structure, Building and Usage instructions to reflect V1.1 changes

[1]PZot Manual Link.

Chapter 2

Structure

2.1 Components

PZotIDE is composed of two main parts: the IDE created using Theia[1] and a Language Server generated using Xtext[2] that takes care of the syntactic analysis of the code.

2.1.1 Language Server

The Language Server protocol is used between a tool (the client) and a language smartness provider (the server) to integrate features like auto complete, go to definition, find all references and alike into the tool.
In this project we implemented a Language Server fot the PZot syntax. The generation of the Language Server is made strating from the PZot.xtex file in the "xtext-pzot-language-server" project of the solution. XText and XTend take care of everything and generate a single jar file that contains the Language Server. This server will communicate through JSON-RPC to our PZot Extension implemented on top of Theia IDE to provide syntactic analysis.
In the "xtext-pzot-extension" project there are also some scripts that copy the generated jar to the extension.

[1]https://www.theia-ide.org/
[2]https://www.eclipse.org/Xtext/

2.1.2 Theia

Theia is an extensible platform to develop full-fledged multi-language Cloud & Desktop IDE-like products with state-of-the-art web technologies. It is designed to work as a native desktop application as well as in the context of a browser and a remote server. To support both situations with a single source, Theia runs in two separate processes. Those processes are called frontend and backend respectively, and they communicate through JSON-RPC messages over WebSockets or REST APIs over HTTP. In the case of Electron, the backend, as well as the frontend, run locally, while in a remote context the backend would run on a remote host.
Both the frontend and backend processes have their dependency injection (DI) container to which extensions can contribute.

2.1.3 PZot Extension

The PZot features of the IDE are provided trough the PZot extension. This allows to update the Theia binary in an efficient way. It is composed by a backend extension that initializes the Language Server and that connects to it and by a frontend component that offers the "user visible" features.
Using the dependency injection the following modules are added to the Theia core:

Backend

- LanguageServerContribution

Frontend

- LanguageClientContribution
- PZotUri
- PZotDependencyGraphOpenHandler
- PZotGraphResourceResolver

LanguageServerContribution

As said before here we take care of starting the Language Server and managing the connection

LanguageClientContribution

This module communicates directly with the text editor component offered by Theia, Monaco Editor. We define the colors of the tokens and some special rules like the allowed parentheses in the file that have a .pzot extension. Here we define also the snippets, portions of code that are automatically suggested to the user when they can be useful. In PZot these allow us not to have to write all the symbols of the original language.

PZotUri

This class is useful to understand when we are actually dealing with a PZot file. Theia can actually open any text file but the features of the PZot extension must be available only for files where they are meaningful.

PZotDependencyGraphOpenHandler

Here we tell Theia that the PZot files can be opened also in another way, other than the textual, using a different widget where we will display the actual graph. When this class gets binded to Theia using the Dependency Injection mechanism a new menu option is added in the "Open With" menu of the PZot files. From a code point of view here we tell Theia what kind of files we can manage with this widget and we have the logic that actually creates the widget (PZotDependencyGraphWidget).

PZotGraphResourceResolver

This is a very simple class that basically creates a PZotGraphResource when needed.
The actual flow is that every time someone opens a PZot file using our custom widget, a PZotGraphResource gets created using the PZotGraphResourceResolver and then is passed to the graph widget.

PZotGraphResource

This class represents the connection between the document and the graph: here we have the events that are triggered when the document changes and the methods to save our changes to the file. We also have many methods that parse the dependency part of the formula.
The parsed formula is stored in a special data structure called pzot-graph that makes use of pzot-node and pzot-edge classes. More details about these can be found in the code documentation in the project.

PZotGraphLayout

Here is the core of the graph representation, where it gets actually rendered. Thanks to the open source library Cytoscape.js we easily represent the graph. All the nodes are structured in columns, each one representing a time "period".
When a new dependency is added to the graph it gets analyzed and the necessary nodes and edges are created. Also the upper and lower bound on the period used by this formula are computed. The idea is to render only the necessary periods in order to ease the visualization. All the modification are reflected also on the PZotGraph data structure that is able to generate the new dependency formula.

Chapter 3

Compiling & Usage

3.1 Requirements

- Node.JS, ¿= 8.9.1, ¡ 9.0.0 (suggested 8.11.3 LTS)
- Yarn Lerna
- Typescript

3.2 Compiling

To compile the solution first we have to build the Language Server whose executable will be copied into the PZot extension.

```
cd xtext-pzot-language-server &&
./gradlew shadowJar &&
cd ..
```

Then we can build the IDE and the extension:

```
yarn
```

We can start the IDE both as an Electron application and as a browser app.

To build and run the Electron app:

```
cd electron-app &&
yarn start
```

To build and run the Browser version:

```
cd browser-app &&
yarn start
```

3.3 Building the Code Documentation

In order to generate the code documentation we make use of Type-Doc[1]. It generates the documentation available in the doc folder starting from the comments in the code.

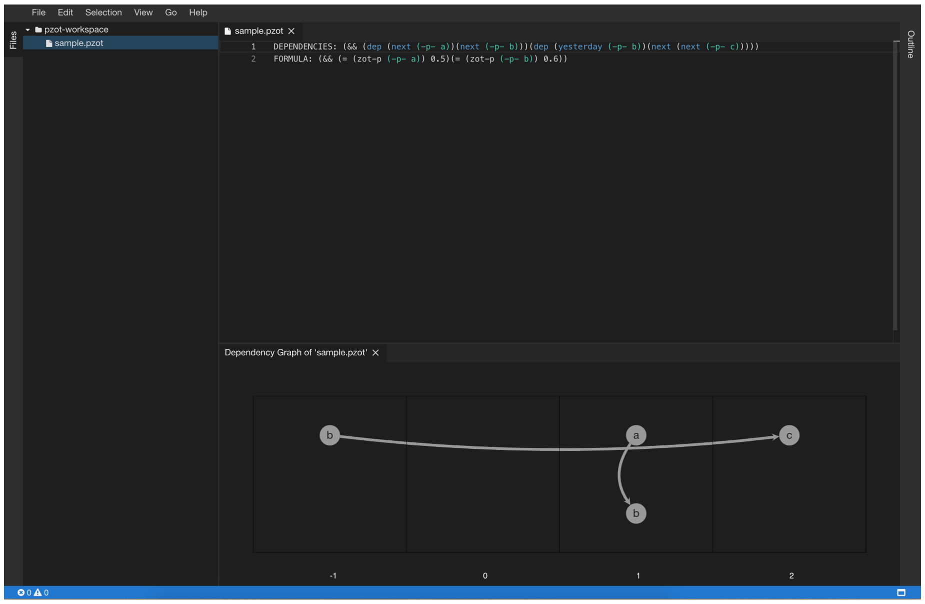
```
yarn doc
```

[1]https://typedoc.org

Chapter 4

Usage

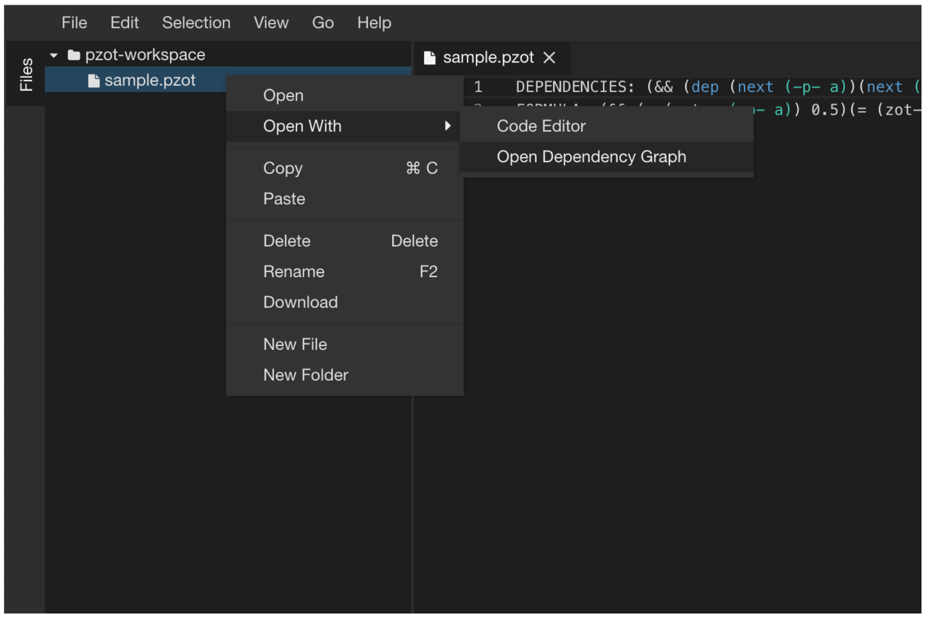
This is the main screen of a PZot file:



```
DEPENDENCIES: (&& (dep (next (-p- a))(next (-p- b)))(dep (yesterday (-p- b))(next (next (-p- c)))))
FORMULA: (&& (= (zot-p (-p- a)) 0.5)(= (zot-p (-p- b)) 0.6))
```

Figure 4.1: Main Screen

On the left side we have the "workspace", a representation of the opened directory with all the files. By clicking the "Files" tab it gets collapsed to give more space to the editor. On the right we have the the text editor on top (Monaco Editor) and the graph widget in the lower part. The widget will appera by right clicking a PZot file and choosing "Open With" and then "Open Dependency Graph".



```
DEPENDENCIES: (&& (dep (next (-p- a))(next (
```

Figure 4.2: Open With Menu

The blu bar at the end of the editor shows the number of errors on the left and the current line and column on right. If we click on the error symbol, the problems panel will open providing us with mode details.

4.1 Text Editor

The text editor makes use of the rules defined by the PZot Extension so it will provide auto-complete, color the tokens and offer snippets of code. Some example of the latter are "and" that inserts the formula: "(&& condition condition)" and automatically moves the cursor to the first "condition". Using the tab key we can efficiently move throughout the various "conditions".
Another really useful snippet is "litteral" that adds all the semantic necessary to declare a litteral.

4.2 Dependecy Graph Editor

Every time we modify the dependency layer of the formula in the Text Editor the graph gets updated. The columns represent the times of the clauses so it may happen to have the same litteral in more columns. We can also modify the dependecy layer by the graph, the following actions are available:

- **Add New Node**
- **Add New Edge**
- **Remove Node**
- **Remove Edge**
- **Move Nodes Between Periods**
- **Compute Formula**

4.2.1 Add New Node

To add a new node we can simply right click on the graph. The new node will have the ”period” of the column were it is created. If is created outside the period grid (left/right) it will add the node in a new (yesterday/next) period.

4.2.2 Add New Edge

To add a new edge we have to hover the pointer on the starting node until a red dot appears. Then we can drag an edge from that dot to the desired node.

4.2.3 Remove Node

To remove a node we just need to right click it. All the edges that belonged to that node are also removed.

4.2.4 Remove Edge

Same procedure as per nodes.

4.2.5 Move Nodes Between Periods

To change the period of a node, we can drag and drop it to desired column.

4.2.6 Compute Formula

Clicking anywhere in the graph will show the Compute Formula button. It will translate the graph into a PZot Dependency Formula and update the document.

Chapter 5

Software & Languages

5.1 Software

- **TheiaIDE** - https://github.com/theia-ide/theia
- **Monaco Editor** - https://microsoft.github.io/monaco-editor/
- **XText** - https://www.eclipse.org/Xtext/
- **Electron** - https://electronjs.org/
- **Cytoscape.js** - http://js.cytoscape.org/
- **TypeDoc** - https://typedoc.org

5.2 Languages

The project makes use of Xtex language for the definition of the grammar of the Language Server and Typescript in the IDE.