# Software Architecture Mining from Source Code Documentation

This documentation describes the installation of the Architecture-Mining project in detail, as well as some basic functionality for its evolution. Please read my master thesis with title “Software Architecture Mining from Source Code” first for a detailed description of the project and its software structure.

## Graph Generator

A tool implemented in C++ that given a C++ project source code extract the dependencies between program classes, builds their dependency graph and store it in a JSON file.

### 1.1 Setup

#### 1.1.1 Third-party Tools

This tool is based on Clang. So, the first step is to download and build the llvm-project (https://github.com/llvm/llvm-project).

For the graph to JSON translation the jsoncpp project is used. To download and install it use the vcpkg dependency manager as defined in project’s readme on github (https://github.com/open-source-parsers/jsoncpp). The vcpkg command for x64 windows is: `vcpkg install jsoncpp:x64-windows`.

#### 1.1.2 Setup Graph Generator on Visual Studio 2019 (VS)

1. Create a VS solution
2. Add all the sources that located in *Architecture-Mining\GraphGenerator* on the created solution
3. Configurate VS solution properties:

- *Debugging > Command Arguments*

  Pass the tools command line arguments (Figure 1)

```
Usage: Architecture_Miner [options] <files...>

Options:
   --src [directory]:    To analyse a whole directory with sources.
                         It can only be used if --cmp-db is not used.
   --cmp-db [file]:      To analyse a system using a compilation database.
                         The file must be named "compile_commands.json".
                         It can only be used if --src is not used.

Files (optional):
   ignored files:        The first parameter is a text file that defines some
                         file paths in order to be exluded from the analysis.
   ignored namespaces:   The second parameter is a text file that defines some
                         namespaces in order to be exluded from the analysis.
   output file:          The last parameter is a JSON file where the generated
                         graph will be stored.
```

*Figure 1 - Graph generator arguments*

- *C/C++ > Additional Include Directories*

  Add the path to the include directories of:

  1. llvm-project
  2. Architecture-Mining
  3. Jsoncpp from *vcpkg\installed\x64-windows\include* path

  See screenshots on Figure 2, Figure 3.

- *Linker > General > Additional Library Directories*

  Add the path where the *.lib* files of llvm-project are located *(path\to\llvm-project\build\Debug\lib).*

  See screenshots on Figure 4.

- *Linker > Input > Additional Dependencies*

  Add the *.lib* files of llvm-project. Add the *version.lib* and *jsoncpp.lib* too.

  See screenshots on Figure 5.

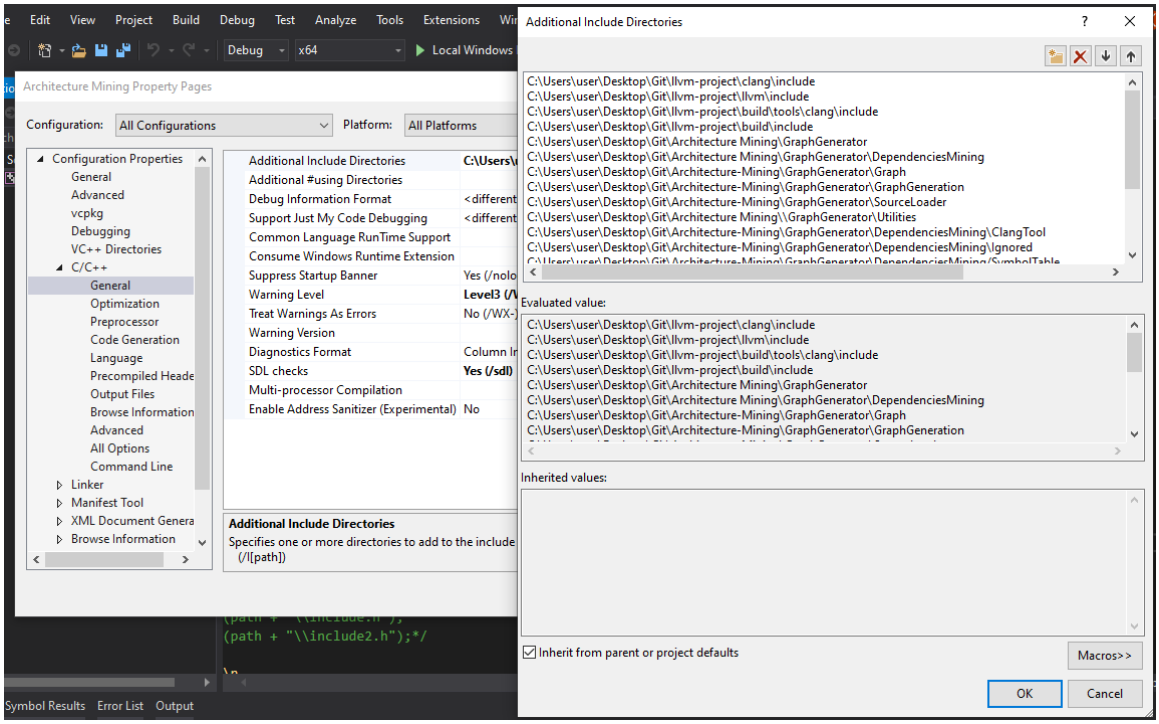


*Figure 2 - Additional include directories on VS (1)*

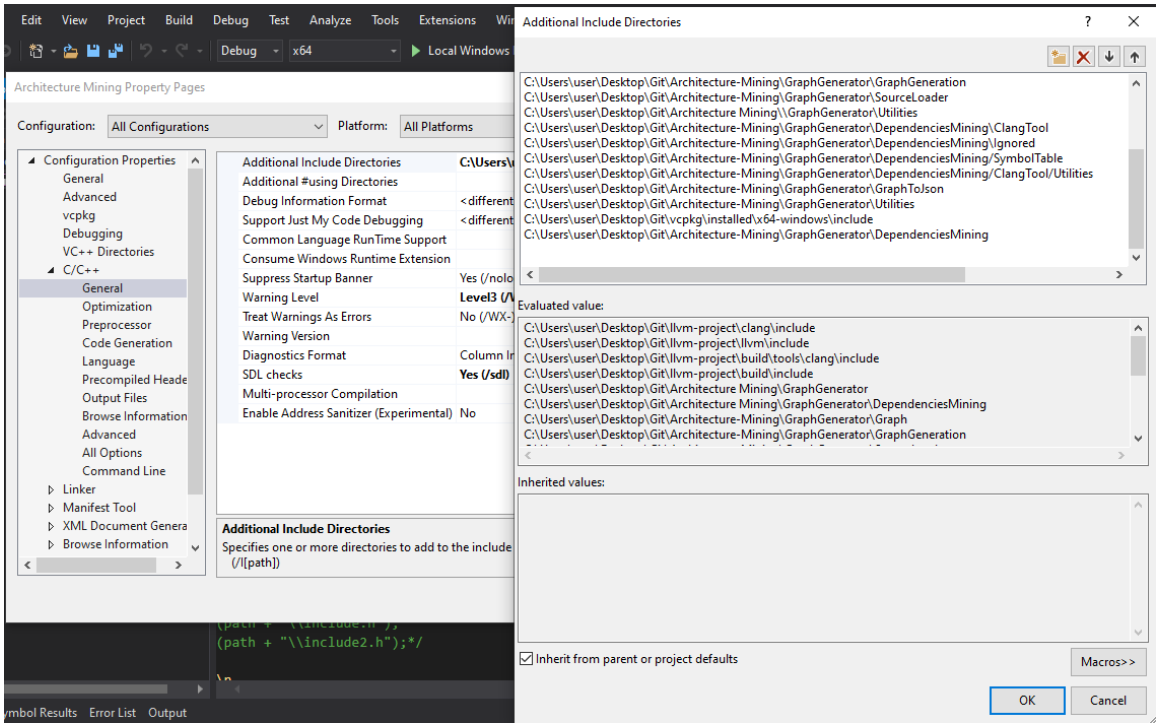


*Figure 3 - Additional include directories on VS (2)*

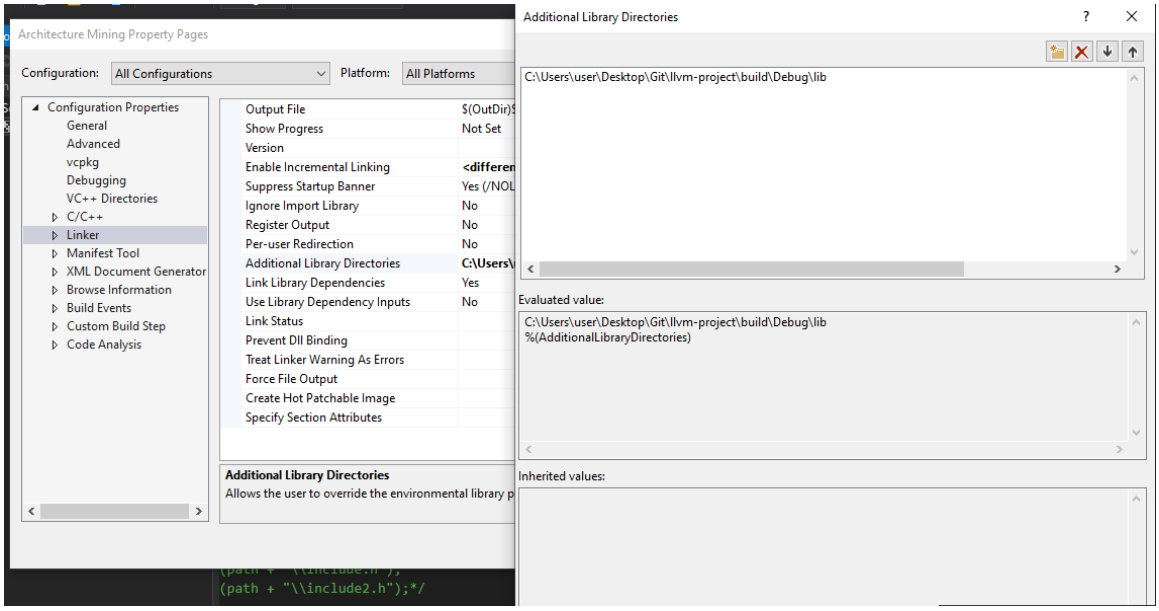


*Figure 4 - Additional library directories on VS*

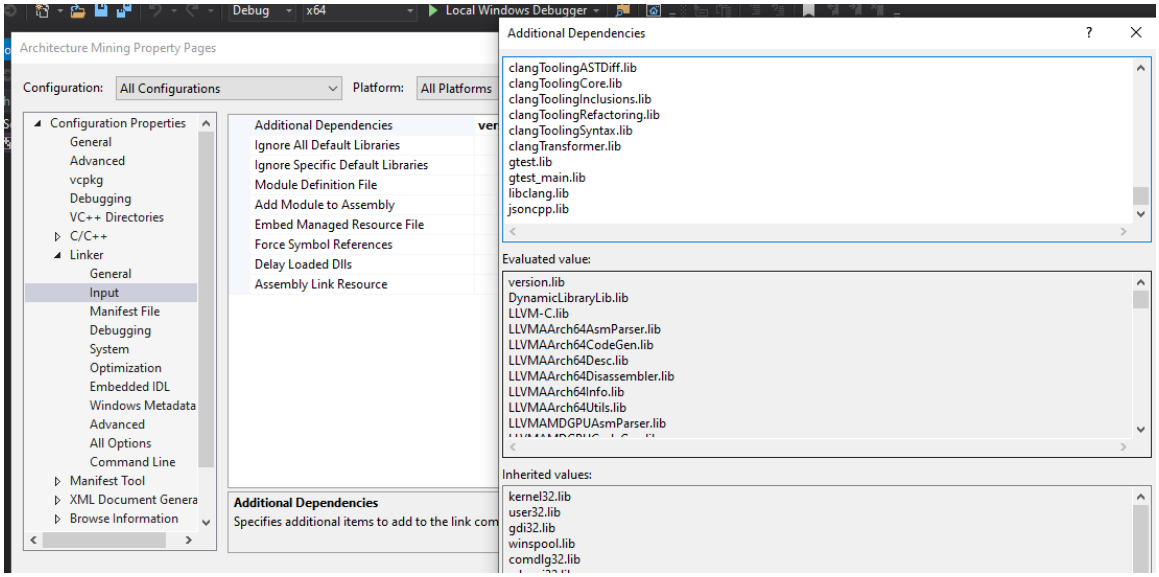


*Figure 5 - Additional dependencies on VS*

## 1.2 Compilation Database

A compilation database is a JSON file that contains an array of “command objects”, each of which specifies how a translation unit is compiled in the project.

It can be generated using cmake, by adding the flag “`-D CMAKE_EXPORT_COMPILE_COMMANDS=ON`.” For example: `cmake -G Ninja -D CMAKE_EXPORT_COMPILE_COMMANDS=ON p*ath/to/project/*`. Only *'Makefile Generators'* or *'Ninja'* generators can create a compilation database. For a Visual Studio projects, it is possible to create a compilation database directly from Visual Studio with the open-source extension *'Sourcetrail'*.

Creating a compilation database will generate a file named compile_commands.json (do not rename this file).

Finally, using a compilation database after changing the source files which it refers to will cause undefined behavior.

# Graph Visualizer

A JavaScript tool that, provided a graph description in standard JSON format, provides its render and interactive configuration.

## 1.2 Setup

Run `npm install` command in the *Architecture-Mining/GraphVisualizer* directory.

## 1.3 Graph Renderer

For graph graphic model creation, we use GoJS library. However, a wrapper over GoJS for graph view interaction and configuration is implemented, in order to communicate with the graph while hiding all GoJS-related details.

Specifically, we export a `graph` object that includes the graph model as well as all of the required functionality in order to interact with it. A list with the provided functionality is depicted in Figure 6. Finally, this `graph` object is defined in *Architecture-Mining/GraphVisualizer/Graph/graph.js* file.

```
        // Get Nodes/Edges Arrays
        getNodesArray,
        getEdgesArray,
        // Get node context menu object
        getNodeContextMenu,
        // Get layouts object
        getLayouts,

        // Add/Remove Nodes/Edges from the graph
        addNode,
        addNodes,
        removeNodeByKey,
        removeNodeByData,
        addEdge,
        addEdges,
        removeEdgeByData,
```

```
        // Find a Node/Node Edges in the graph
        findNode,
        findNodeEdges,

        // Insert a node in a group
        // groupKey == undefined if it has no group
        setGroupToNodeByKey,
        setGroupToNodeByData,
        getGroupOfNodeByKey,
        getGroupOfNodeByData,
        nodeIsGroupByKey,
        nodeIsGroupByData,

        // Hide/Show Node/Edge
        hideNodeByKey,
        hideNodeByData,
        showNodeByKey,
        showNodeByData,
        hideEdgeByData,
        showEdgeByData,

        // Hide/Show Edge Weight
        hideEdgeWeightByData,
        showEdgeWeightByData,

        // Set Node/Edge Color
        setNodeColorByKey,
        setNodeColorByData,
        setEdgeColorByData,

        // Group/Outer Layout
        setGroupLayout,
        setOuterLayout,

        // nodeContextMenu
        nodeContextMenuAddProperty
```

*Figure 6 - Graph API*

## 1.4 Graph Configuration Panel

To explore and manipulate the generated graph, we introduce a configuration panel that allows interactive changes on graph. This panel is dynamically constructed, based on a

JSON file that describes its structure. Section 6.2.2 of my thesis contains details about how to form and use it. You may also look at the JSON Schema (*Architecture-Mining/GraphVisualizer/Config/ config_schema.json*) that is defined for the validity check to get a better understanding of its structure.

## 1.5 Add a New Graph Configuration

The communication between graph renderer and interactive configurator is succeed using an observer (read section 6.2 from my thesis). So, in order to install a new configuration, you have to add a configuration control element in configuration panel in order to manipulate it as well as install the event handler that will apply the configuration on graph.

### 1.5.1 Add a Configuration on Panel

To add a new configuration control element on this panel, you have to add its description on *Architecture-Mining/GraphVisualizer/Config/config.json* file (in this file it is described the current configuration panel). The rules regarding its structure are defined in Architecture-*Mining/GraphVisualizer/Config/ config_schema.json* file.

### 1.5.2 Install a Configuration Callback

To apply a configuration on graph, you have to define and install appropriate event handler, which will be fired when the event occurs. To accomplish this, we implement `configApplicator`, which provides all functionality for installing and uninstalling event handlers, as well as storing globally some values that can be shared among event handlers. In addition, a callback function that will be invoked when an event handler action completes can be specified. The main use of this callback is to allow a configuration to interact with other configurations by using the previously mentioned global values. All of them are defined in *Architecture-Mining/GraphVisualizer/Graph/configApplicator.js.*

It is notable that in *Architecture-Mining/GraphVisualizer/Config/config.json* you have to link the control element with the event handler using the unique id that is defined at handler installation in order to be fired when the event happens.

In the following section, we will include an example to help clarify the technique discussed above.

### 1.5.3 Example

In this example we define the weight filter. In Figure 8 its description in *config.json* file is depicted that generated the slider in the configuration panel as represented in Figure 7.

For the event handler we define the event handler function and install it in observer (Figure 9). In order to be fired when the weight filter changes, it is required to link the filter control element with the event handler using the unique id that is defined at installation (highlighted in Figure 8 and Figure 9).

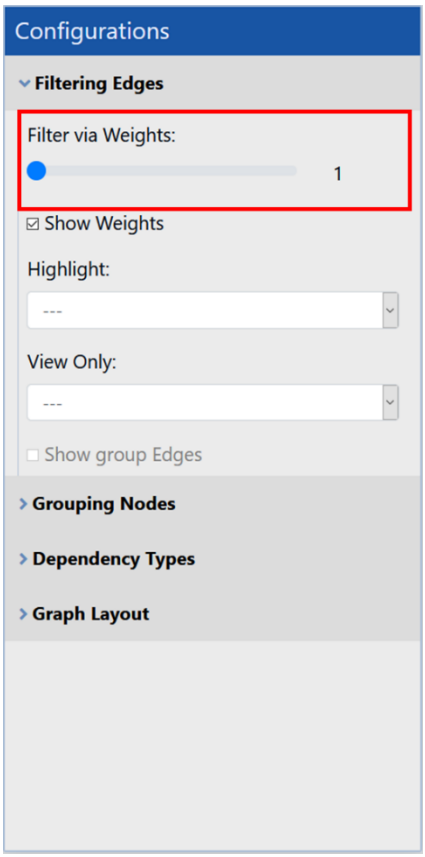


*Figure 7 - Weight filter in configuration bar*

```
"weightFilter": {
    "id": "weightFilter",
    "description": "Filter via Weights",
    "type": "slider",
    "value": 1,
    "min": 1,
    "max": 1000,
    "step": 0.5,
    "conditions": {},
    "onChange": {
        "event": "weightFilter"
    },
    ...
}
```

*Figure 8 - Weight filter description in config.js*

```
function weightFilterHandler(value) {
  // ...
}

configApplicator.install('weightFilter', weightFilterHandler);
```

*Figure 9 - Weight filter event handler definition and installation*

## 1.6 Run

In the *Architecture-Mining/GraphVisualizer* directory run *npx serve* command and follow the link to direct you to the graph visualizer web app.