

SOFTWARE MAINTENANCE AND EVOLUTION

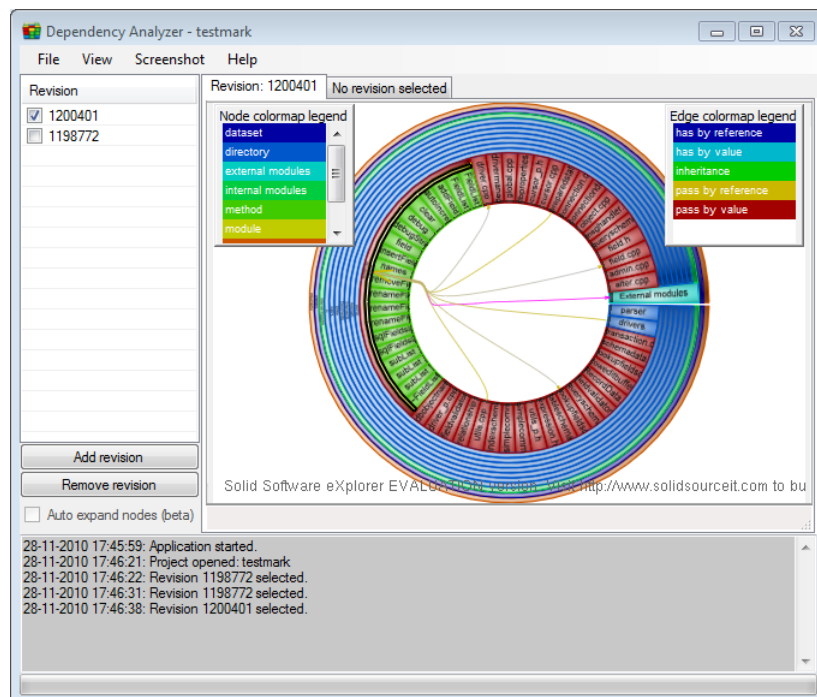
Project: Dependency Analyzer

Students:

Mark Ettema (S1922017)
Erwin Vast (S1924451)

Lecturer:

Prof. Alex Telea



November 30, 2010

Contents

1	Introduction	5
2	Requirements	7
3	Solution methodology	9
3.1	Create a new DA project and retrieve source code	11
3.2	Extract dependencies and generate XML reports	11
3.3	Parse XML reports	14
3.4	Generate database for SolidSX	14
3.5	Visualize dependencies	17
4	Results	18
4.1	Example projects	19
4.2	Demonstration video	19
4.3	Source Files	19
4.4	CCCC quality	19
4.5	File parse problem with KDOffice	20
5	Conclusion	21
A	Developer manual	23
A.1	Building Dependency Analyzer	23
A.2	Building CCCC	24

1 Introduction

The lifecycle of a software development project typically consists of a design, development, test and maintenance phase (mostly iterative). The majority of the budget for software projects is devoted to maintenance. Around half of this budget is just for understanding the software. So tools that make it easier to understand the software can greatly reduce maintenance costs.

There are four different kinds of maintenance:

- **corrective maintenance:** fixing bugs after they are observed.
- **adaptive maintenance:** implementation of new or changed requirements.
- **perfective maintenance:** improve the quality of the software (e.g. refactoring, improving availability and performance).
- **preventive maintenance:** fixing causes of defects in advance of failures.

To do perfective maintenance, one needs to know the dependencies of a software module. A software module can, for example, be a code file (C) or a class (Java or C#). If the dependencies of a module are (partially) unknown for a developer, this can lead to defects in the software.

For example, suppose a project uses an external library and a new version of this library changed an assumption about the input parameters of a function. If not all the dependencies are known for the developer, it is possible that the change is not implemented in all the project code.

So, to state the problem: *How can we make it easier for the developer to identify the dependencies of a software module?*

During this project, a development tool is created that can visualize the dependencies of a software module and the evolution of the dependencies. This way, the developer can check the dependencies of a module, without even looking at the code! And because the evolution of the dependencies are known, a developer can also see which dependencies have *changed* compared with another version of the project.

In the next chapter, the requirements of this tool are stated. These requirements lead to the solution methodology in chapter 3, where the implementation of the developed tool is explained in detail. The results of the tool are described in chapter 4, which contains some screens of the tool and a discussion of the quality of the tool. The last chapter, chapter 5, contains a summary of the project and a discussion how the tool could be improved.

2 Requirements

A part of the development of the tool is to clearly state the requirements. Overall, the tool should visualize the dependencies of the modules and the evolution of these dependencies. The high level requirements are:

- The tool can extract dependencies of a C/C++ code project.
- The tool can visualize these dependencies for every module (dependency edges between the modules).
- The tool can retrieve the source code by a local directory.
- The tool can retrieve the source code using the subversion protocol.
- The tool can save the project's dependencies so that the dependencies can be watched at a later time.
- The user only has to use the developed tool and is not aware that other tools do part of the extraction process.

How the dependencies are visualized is not a requirement at this stage. There are different ways to visualize the dependencies and they could all be valid solutions. Therefore, in the next chapter, the decision which technique to use is discussed.

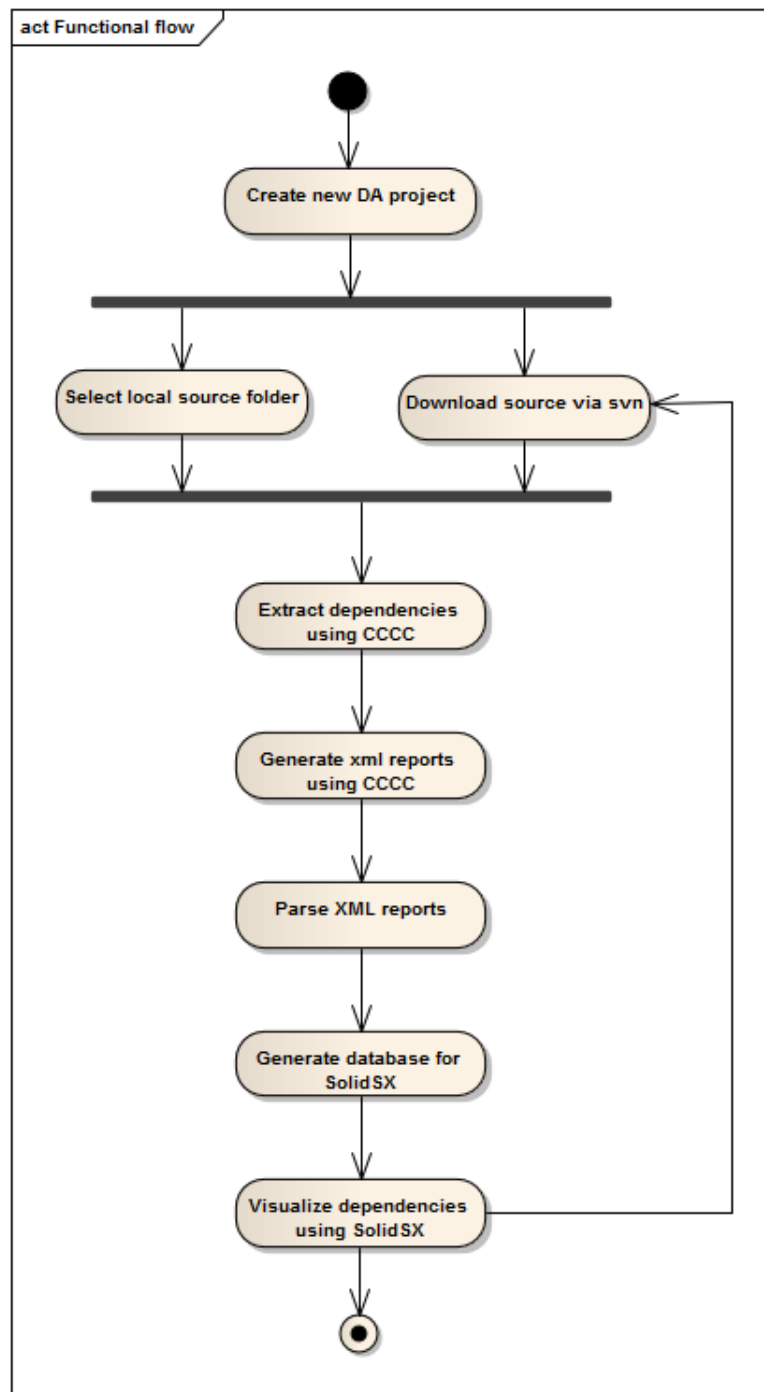


Figure 1: Activity diagram showing the high-level functional steps.
Note: here is already chosen for SolidSX for visualizing the dependencies for the simplicity of this document. The rational for this choice will be explained later on in chapter 3.

3 Solution methodology

In the previous chapter, the high-level requirements were listed. In this chapter, the solution for realizing this is described. This section describes the design of the software and describes the steps to retrieve and visualize the data, the data structures and which algorithms were used.

To start with the source code and finish with the evolution of the dependencies is not a trivial subject. Therefore, this section describes in detail which steps are taken from downloading the sourcecode to visualizing the evolution of the dependencies.

Figure 1 on the left page shows an activity diagram with the high-level steps that are performed in the software. In the next sections, each of these steps are described in detail. Also, the choice for certain programs, like SolidSX and CCCC will be argued in their corresponding sections. Below is a description of the diagram.

First a new project file is created for the dependency extraction. This makes it possible to save the dependencies and visualize it again in the future, without doing the extraction procedure again. The second step is to access the source code either selecting a local folder, or downloading the source code given a link to a subversion repository. When subversion is used, more revisions can be downloaded at a later time. The next step is to extract the dependencies. To extract the dependencies, the C and C++ Code Counter (CCCC) is used which can be downloaded from <http://cccc.sourceforge.net/>. This tool extracts the dependencies from the source code and in the next step, it saves the dependencies into files using the XML format. CCCC also extracts the declared modules (classes) and functions. The fifth step is to extract all this information from the XML files to C# Object Oriented modules in memory. This makes it possible to create a SQLite database in the next step that is supported by SolidSX and which can, finally, visualize the dependencies in a nice graphical way.

Note the arrow from the last activity to the download source activity. This visualizes that when using a subversion repository, other versions of the source code can be downloaded and extracted to show the evolution of the dependencies between the different versions.

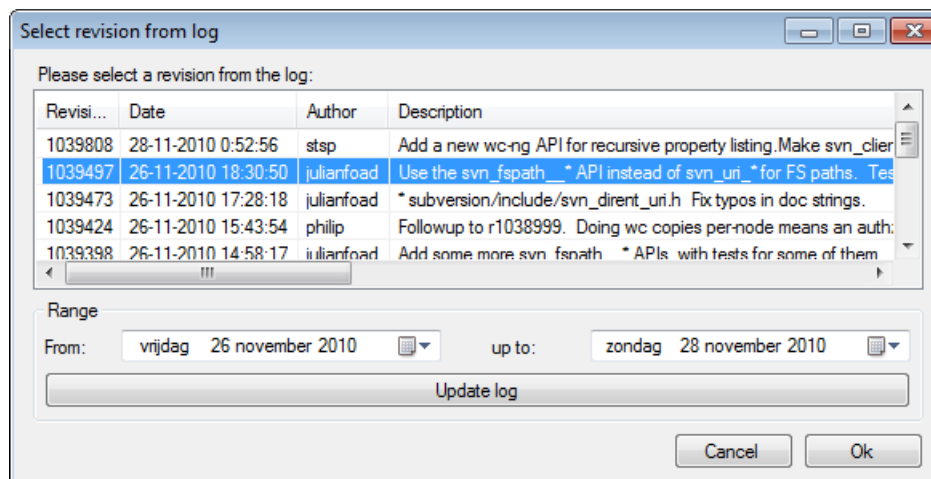


Figure 2: Selecting the right revision using the svn version log.

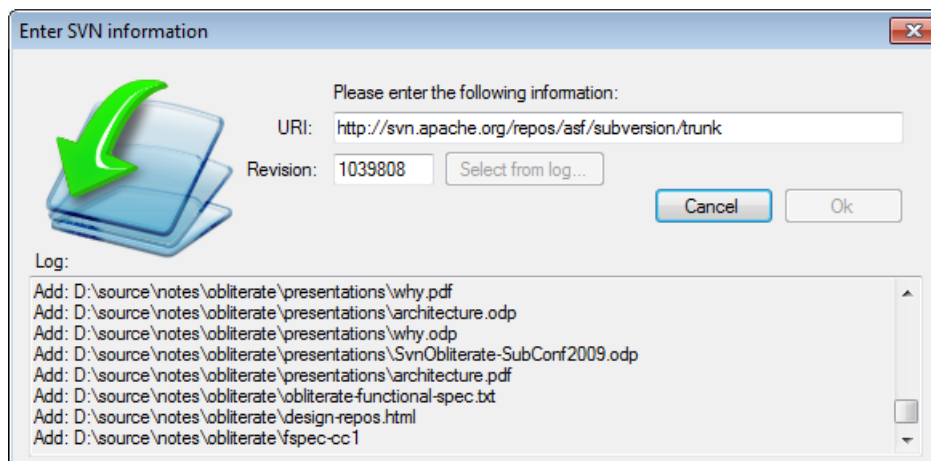


Figure 3: Doing a checkout or update using svn.

3.1 Create a new DA project and retrieve source code

The first step of the process is to create a new Dependency Analyzer (DA) project containing the information of the DA project. Each DA project consist of a directory containing a project file and three subdirectories:

- .cccc: Contains the XML reports generated by CCCC.
- dependencies: Contains the dependency files for the DA tool.
- source: The sourcecode from the latest downloaded svn revision.

The project file (XML) stores the DA project important information and has the following structure:

- The project creation date.
- The project name.
- Whether a local directory or svn repository is used.
- In case of svn: the link to the repository and the revision numbers of the revisions for which the dependencies where extracted.

The next step is to retrieve the source code. The first option is to extract the dependencies for a project on the local disk. This has the advantage that the source code should not be downloaded again, because it is already on the local disk. A disadvantage is that no new versions of the code can be downloaded using SVN. (This could be an improvement of the software. If SVN directories are already in the local folder, it could also be used for retrieving new versions of the software).

Therefore, there is also the possibility to retrieve the source code using the subversion protocol. To use the SVN protocol in the DA tool, the SharpSvn library is used, which can be found at <http://sharpsvn.open.collab.net/>. Using the version log of the different versions, the user can select the appropriate revision from the log windows (see figure 2). After the appropriate version is selected, this version is downloaded from the SVN repository (see figure 3).

3.2 Extract dependencies and generate XML reports

After the source code has been acquired, the next step is to extract the dependencies from the source code. For this task, different tools are available, like C and C++ Code Counter (CCCC) and Rigi. Both tools were tested, but CCCC is used, because CCCC worked out of the box and it was not possible to get Rigi working (Rigi is also a very old project).

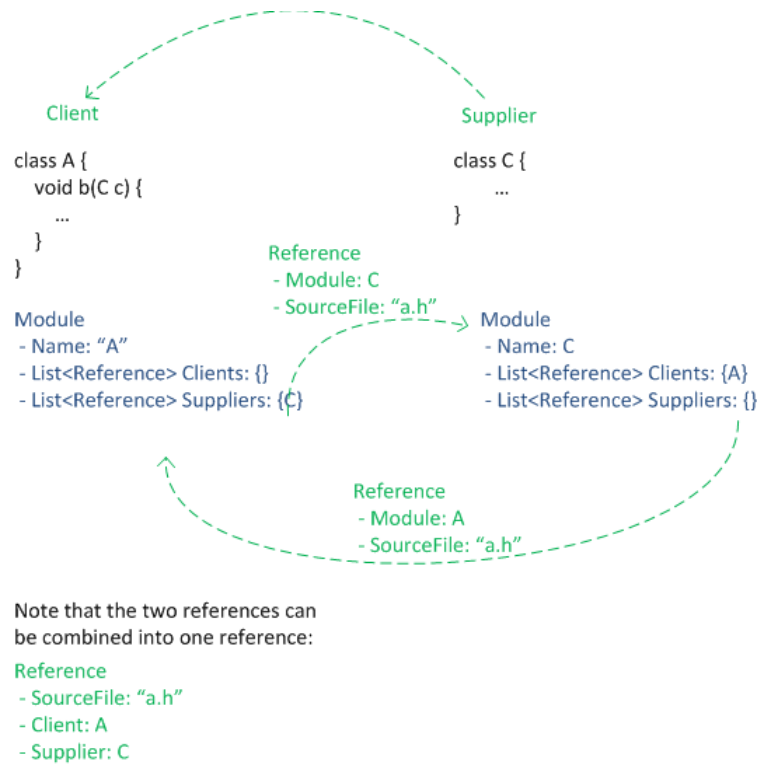


Figure 4: Diagram illustrating the meaning of a created XML report.

CCCC is a tool that can extract different kinds of dependencies and stores the dependencies in XML files for each module. Unfortunately, CCCC does not support extracting call dependencies, but it can extract the following dependencies from the source code.

- pass by value
- pass by reference
- has by value
- has by reference
- inheritance

Each XML rapport contains the following information for each module:

- Code information: e.g. McCabe's cyclomatic complexity, lines of code, inheritance level.
- Module information: module declarations and definitions.
- Procedural detail: module function declarations and definitions.
- Structural detail: module dependencies.

To visualize the dependencies, the structural detail is the most interesting part. The structural detail consist of the dependencies per module and it split in client modules and supplier modules. Each module lists the following information:

- Modulename
- Visible (true/false)
- Concrete (true/false)
- For every dependency:
 - The kind of dependency (e.g. inheritance, pass by value)
 - The location of the dependency (e.g. source file and line number)

To process the reports, it is important to understand the rapports (semantics). Just parsing the reports will not help to visualize the dependencies. What are exactly suppliers and clients, and are the location of dependencies the location of the usage or the location of the declaration of the value? Therefore, some analysis has been done and figure 4 shows the result of this analysis.

In figure 4 you can see an example scenario with two classes, class A and class C. Class A has a function that has a (pass by value) dependency to class C, because the type C is used as paramater type for function b. In that case, class A is the client and class C is the supplier. CCCC creates for both classes a report, suppose A.xml and C.xml. Both XML files have the dependency, but in one XML file as client and in the other file as supplier.

The XML files are presented as the modules in figure 4. This gives a more object oriented impressions of the files, but contains in essence the same contents as the XML files. Using object oriented way, it gives a start for parsing the XML into C# objects.

In the example scenario, the left module has a reference to module B and the right module has a reference to module A. Note that both references contain the same sourcefile: this is thus always the location of the usage and not the declaration of the object! This is important to know, because now the sourcefile of module C is unknown based on the reports. Also note that in C#, the references could be combined to one reference, as displayed in the figure.

Now the meaning of the reports are clear, the next step is to process the reports into C# modules so it can be further processed. This is described in the next section.

3.3 Parse XML reports

The next step, after the dependency extraction is to parse the created XML files and store the information in C# objects in memory. The parsing consists of the following steps for each file:

1. The module name is extracted.
2. The function declarations are extracted.
3. The references are extracted, containing the type of reference, source file and source line.

The results of this extraction are stored in the class *Module*. The following step is to link the module to the other modules. Remember the client - supplier relationship that has to be stored, so this relationship is stored in the class *Reference*. The module has to be linked to the reference and vice versa for both client and supplier.

The last step is to remove the modules without dependencies. This ofcourse needs some clarification because why is then the module stored as XML file? This is because CCCC creates an XML file for each module it encounters, even if it has no dependencies. But the file is parsed and included in the list of modules in memory, because the methods are extracted. To prevent that nodes are visible in the visualization but do not have any dependencies (and thus are not interesting) are removed from the list of modules in memory.

Now all the modules are linked and stored in memory, the next step is to visualize this data!

3.4 Generate database for SolidSX

To visualize the dependencies there are different kind of libraries that could be used for this. At first, Microsoft Chart was used. But it was difficult to create the custom diagrams and it was also possible to integrate SolidSX into the DA application that could create the graphics. Therefore, the SolidSX visualization component was used and integrated in the DA application. SolidSX has to know the following three points to visualize the dependencies:

- Nodes: the name and id of each node.
- Hierarchy: the parent and child nodes to visualize the directory tree structure.
- Edges: the edges (dependencies) between the nodes with their kind of dependency (e.g. inheritance, pass by value).

Since it is not trivial to go from the one dimensional list to the two dimensional tree structure that is needed for the visualization, this will be described in more detail.

To create a directory tree for SolidSX from the one dimensional list, a tree structure is build in memory. This has as advantage that it is similar to the directory structure of the files and it is faster to search for nodes (compared to creating a similar list in one dimension). Once every node is added to that tree, the tree is stored in the database for SolidSX.

The steps to create the database with the dependencies for SolidSX are listed below.

1. Four default, top-level nodes are created: project name, dataset, internal and external nodes.
2. All the modules are added to the tree.
3. For every module, their methods are added as nodes.
4. Every directory name of the full location of the node (source file) is added to the tree as nodes.
5. Every sourcefile is linked to the right directory node.
6. Every module is linked to the right sourcefile.
7. Every method is linked to the right module.
8. Every directory is recursively linked to each other.
9. Every edge (dependency) between the modules is added.

Perhaps when reading the first point, the question arises what internal and external nodes are. Remember from the previous section that CCCC stores an XML for every module it encounters. So even when the modules are not declared in the source files scanned (because the source code is not complete or of scanning errors), it is stored in XML (but without the source code information). As a result, a certain amount of modules have dependencies to each other, but one or both sides of the dependency is not included in the source code. Therefore, these modules are contained under the node for external nodes. These files can not be added to the internal nodes, because the source files are not present and thus have no location that can be used to store the file in the directory tree.

Another point that needs more clarification is the fifth point. In the previous section it was argued that the sourcefile of the scanned module is not stored in the XML file. How is then the location of the sourcefile retrieved? To get this location, the CCCC code was altered in such a way that the location of the scanned file is added to the XML file. This has an advantage that the source file location can be shown when hovering over the nodes.

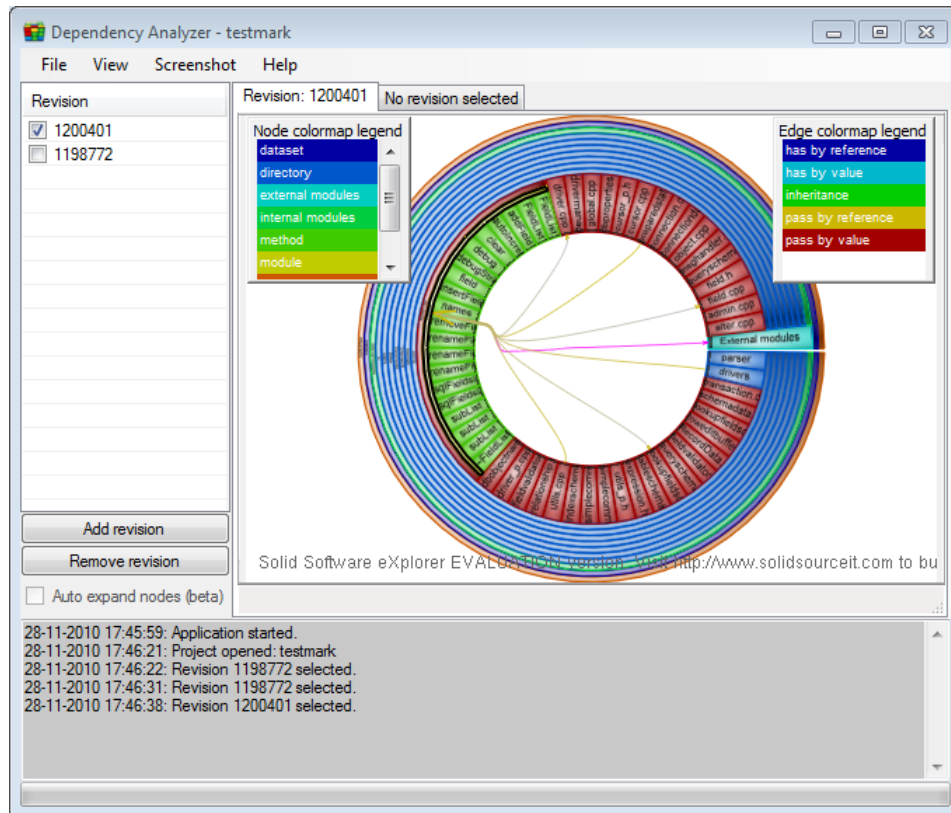


Figure 5: Example visualization of the dependencies in DA using SolidSX.

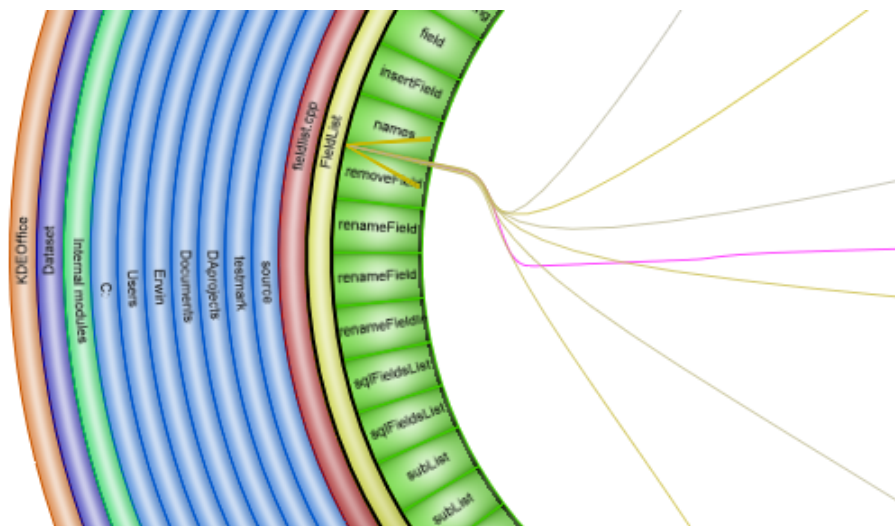


Figure 6: Zoomed in at the selected node.

There are two ways to visualize the dependencies in SolidSX, using XML and using SQLite. At first, XML was used to create the database containing all the information that SolidSX needs. But it was not very fast, so also an SQLite variant was developed. This was a great success, because when using SQLite, the loading of SolidSX was almost three times faster! Therefore, SQLite is used to store the dependencies, but the XML generator is still available in the source code (but not used).

3.5 Visualize dependencies

Now the database is created, the dependencies can be visualized using SolidSX in DA (see figure 5). The left side of the application shows the revisions that were downloaded and processed. Under the list of revisions is the possibility to add more revisions from the repository. On the right side the dependencies are visible from the selected revision. How does it work?

In the center is the main ring structure, where each ring matches to one level in the directory hierarchy (except for upper default rings). This is good visible when zoomed into the selected node as displayed in figure 6. Here the hierarchy is good visible, where the path of the sourcefile is `C://Users/Erwin/Documents/DAProjects/testmark/source/fieldset.cpp`.

There are different colours for different kind of nodes e.g. directories, files, modules, methods. There are also different colours for different kind of edges e.g. has by reference, inheritance, pass by value. The pink edges are edges which combine edges of different kinds.

It is also possible to compare dependencies of different versions of the source-code. Therefore another version in the revisions list can be selected, which opens SolidSX in the second tab in the right panel. This way, the evolution of the dependencies can be easily analyzed.

Another button on the left side is the *Auto expand nodes (beta)* button. This button enabled an experimental features that makes it possible to open a node on the left tab which is then automatic opened on the right tab (or vice versa). This makes it possible to easily navigate and compare dependencies in the rings on both tabs. This comes to the idea to display the rings side by side, which would make it even easier to compare, but because of the size of the ring and the limited size on most displays, this was not implemented.

To explain how the dependencies can be further analyzed and how to detect changes in the dependencies, it is more easy to explain this in a demonstration than in text. Therefore, a demonstration of this functionality is included in the demonstration video which is included on the CD.

4 Results

The result of the project is an easy to use application which can analyze and show dependencies of C/C++ code. The code to analyze can be in a local folder or a SVN repository. A complete description of how the software should be used is described in the "Dependency Analyzer: User Manual". In the user manual document is also described how the software should be installed and how it can be removed from the system.

Next to the software installations, which is described in the user manual, the CD also contains some other files. The CD have the following folder structure:

1. *AutoPlay*: This folder contains some files necessary for the CD menu.
2. *DAProjects*: This folder contains some example projects.
3. *DemoVideo*: This folder contains a demonstration video which is available in several codecs.
4. *Documentation*: This folder contains a PDF version of the report and the user manual.
5. *Setup*: This folder contains the setup files for Dependency Analyzer.
6. *SourceFiles*: This folder contains the source code of Denpendency Analyzer and CCCC.

When the CD is inserted into the computer the CD menu appears automatically (see figure 7). Click on "Browse CD..." to explorer the CD and see the folder structure.

Note: If autoplay is not enabled, navigate to the CD and start Dependency-AnalyzerMenu.exe manually.

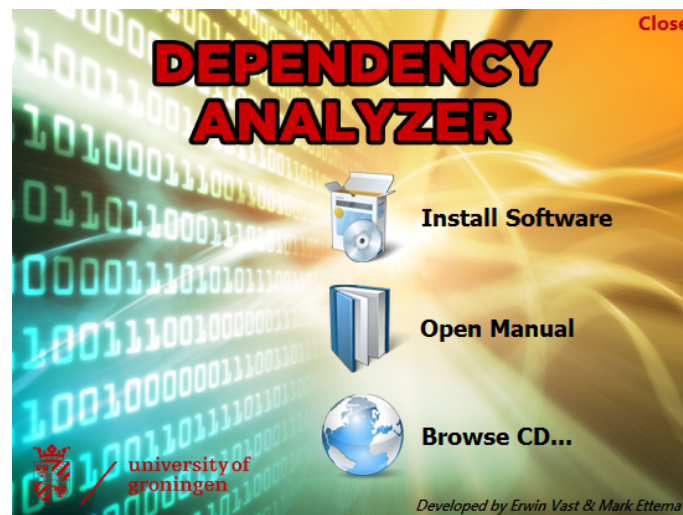


Figure 7: *Dependency Analyser CD menu.*

4.1 Example projects

In the *DAProjects* folder on the CD two example projects can be found. A small one *KexiDB* and a large project *KDOffice-svn*. Both projects are archived using ZIP. To open the projects unzip the project archive to a location on the computer. After that it is possible to open the project with Dependency Analyzer, in the user manual is described how this works.

4.2 Demonstration video

To make it easy to see what the software can do we have made a demonstration video, which shows nearly all features of the software. To be sure it is possible to open the video on several systems it is available in the following formats:

- Audio-Video Interleaved (.avi)
- MPEG-4 AVC Video Format (.mp4)
- MPEG-2 Movie (.vob)
- Windows Media video (.wmv)

The videos can be found in the *DemoVideo* folder on the root of the CD.

4.3 Source Files

The *SourceFiles* folder on the CD contains the Dependency Analyzer Visual studio 2010 solution and the CCCC source. A description of how these files can be used can be found in the "Developer manual" which is an appendix of this document.

4.4 CCCC quality

During the usage of CCCC for the developed tool, it became clear that CCCC is not a perfect tool, because it prints a lot of error messages during the extraction of the dependencies.

When CCCC encounters modules that are undeclared because of missing code, it fails to scan the rest of the file. Thus missing code decreases the quality of the extraction significantly.

Another problem is that CCCC does not support certain C++ syntaxes. An example of this are words between the class keyword and the class name, like *class KFORMULAPRIVATE.EXPORT KFormulaPartDocument*. CCCC fails to parse these kind of syntaxes. We actually tried to fix this problem, which was not easy because the CCCC source code is rather complex. We removed every word between the class keyword and class name. Initially, this seem to work, but the change actually introduced new failures during

parsing. This was caused by a lot of exceptions of the problem case, like forward declarations or the word *class* in the source comments.

The fix was actually not furthered used, because it did not really improve the software because it introduced new failures. But it shows that changing the CCCC source code (and parsing code in general) is a complex task which is not easy to do in a short time.

4.5 File parse problem with KDOffice

When we used KDOffice to test our application we had the problem that there was one file that could not be parsed by CCCC. Because CCCC get stuck it did not finish the extraction of all dependencies. Our first "solution" was a hard coded ignore list in which we added the file that caused the problems. In this case KDOffice can be parsed well, but what about other source code with the same problem? Because we did not like the first solution we have checked the reason why CCCC get stuck. CCCC read the files until EOF, but in the infamous file there was a non ASCII char which gives the problems (CCCC reads it as an EOF). To solve this problem we build in a check to skip the non-ASCII chars. This seems to work, but it makes CCCC very slow, so for that reason we did not use it.

The final solution works like a watchdog inside a loop. It is a counter which counts up to a specified value, when this value is reached the file will be skipped. We have tested this solution with the KDOffice repository and it works well.

5 Conclusion

The challenge of the project was to develop a program which can extract dependencies of C/C++ code and show this (big amount of data) on a well-organized way to the user. It should be possible to enter directly a subversion uri and it should be possible to compare the analysed revisions. All these requirements should be integrated into one application.

We have used SharpSVN to give the user the possibility to directly enter a subversion uri, an optimized version of CCCC is used to extract the dependencies. Intergrating SolidSX is done to give a nice visualisation of the data, where the two tab view makes it easy to compare serveral revision. To improve the compare part of the software more the experimental "Auto expand node" function is developed.

During the project we have also noticed some possible improvements for the future.

- Selecting multiple revisions when using a subversion project.
- Optimize the software so that it can calculate the dependencies by only parsing the changed files. This makes the calculations process faster.
- Showing also the function call dependencies.
- Optimize the "Auto expand node" feature when there are more events from SolidSX available. A mouse listener is not needed anymore and with more event types it is also possible to collapse or recursively expand the nodes in both tabs at the same time.

Because of the time it was not possible to implement these features already.

In short it was a nice and interesting project where we have learned a lot. Not only about extracting dependencies, but also about combining several tools and integrating them into one user friendly application. Of course we have considered some points of improvements, but we had made decisions in order to the time. We have enjoyed working on the project and are pleased with the result.

A Developer manual

This appendix describes how to build the software using the source code from the *SourceCode* directory of the CD.

A.1 Building Dependency Analyzer

Dependency Analyzer is developed in C# .NET using Microsoft Visual Studio 2010. The complete solution can be found on the CD in the folder: *CD:\SourceCode\ DependencyAnalyzer*. The solution file contains two projects: The Dependency Analyzer and the installer. Now it is quite easy to build the software, just open the solution and choose for build. When there is something changed in the Dependency Analyzer, just rebuild the installer and the installer is also up-to-date. Because the program uses external executables we made a folder *Output* which contains the right version of *cccc.exe* and *sqlite3.exe*. When a build is done, the file from this directory will be copied to the *bin\debug* or *bin\release* folder.

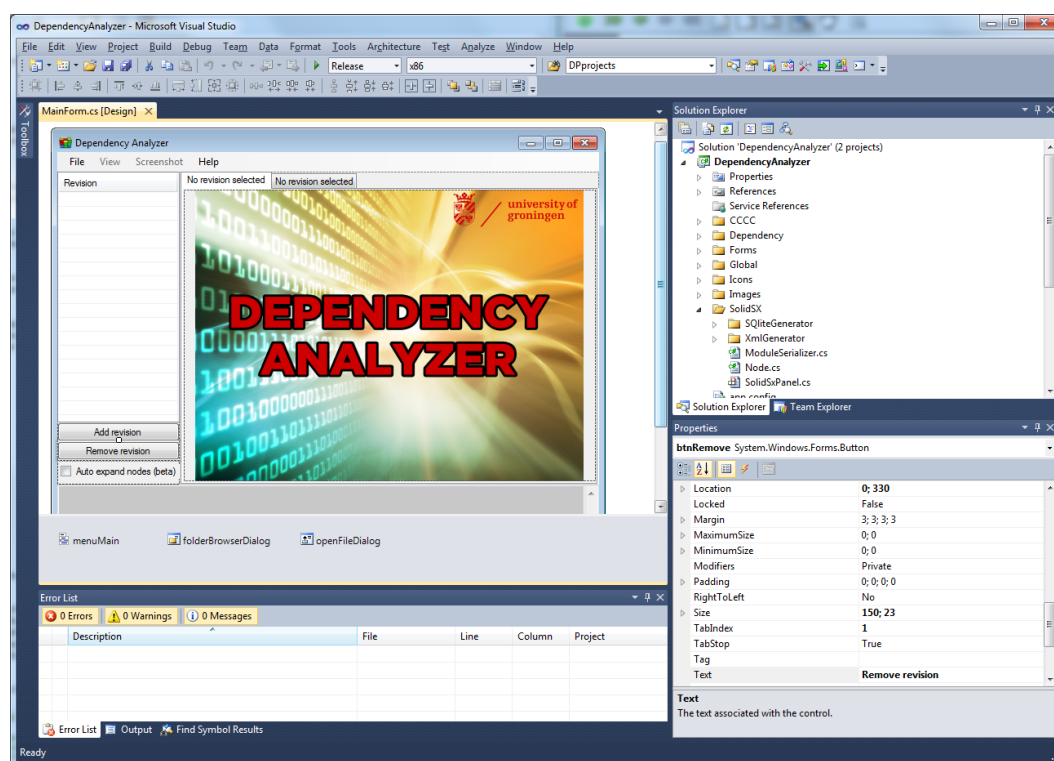


Figure 8: *Dependency Analyzer Visual Studio 2010 Solution.*

A.2 Building CCCC

The CCCC source code can be found on the CD in the folder *CD:\SourceCode\cccc-3.1.4*. To build CCCC, first download the Purdue Compiler Construction Tool Set from: <http://www.polhode.com/pccts133mr.html> and extract it to the CCCC source folder. Then add the Visual Studio path to PATH in the commandline (cmd): *set PATH=C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE;%PATH%*. Then execute *build_w32vct2003.bat* in the commandline. If an error occurs that Kernel32.Lib is missing, copy Kernel32.Lib from the Windows SDK to the cccc folder, like *C:\Program Files\Microsoft SDKs\Windows\v7.1\Lib*.

To build the software after a change to CCCC, first execute the command *build_w32vct2003.bat --clean*, then extract the Purdue Compiler Construction Tool Set and then execute *build_w32vct2003.bat*. The executable is created in the cccc directory.