VizzAnalyzer C/C++ Front-End
Using Eclipse CDT

Xuan Wang
Bachelor Thesis

VizzAnalyzer C/C++ Front-End
Using Eclipse CDT

Xuan Wang

May 2009

Department of Computer Science
Växjö University

Supervisor:
Rüdiger Lincke, Phil. Lic
Abstract

VizzAnalyzer is a stand-alone tool for analyzing and visualizing the structures of large software systems. Currently, it only supports analyzing Java and UML programs. Considering the widespread acceptance of C/C++ program languages, we think it is necessary to create a C/C++ Front-End to enable VizzAnalyzer to analyze C/C++ programs.

To create the C/C++ Front-End, we need to get C/C++ Front-End Meta-Model first. For doing this, we selected Eclipse CDT as the compiler for C/C++ source files. Secondly, we create a mapping between C/C++ Front-End Meta-Model to Common-Meta-Model. The mapping result will be used by VizzAnalyzer to do further analysis work.

This Bachelor thesis documents the relevant theory to this C/C++ Front-End and how it has been developed and implemented.

Keywords: Eclipse CDT, C/C++ Front-End Meta-Model, VizzAnalyzer, Common-Meta-Model
Table of Contents

1. Introduction ................................................................................................... 1
   1.1 Problem..................................................................................................... 1
   1.2 Goals and Criteria ..................................................................................... 1
   1.3 Motivation................................................................................................. 2
   1.4 Outline ...................................................................................................... 2
2. Background ................................................................................................... 3
   2.1 Program Analysis Process ......................................................................... 3
   2.2 Meta-Model Theory .................................................................................. 4
   2.3 VizzAnalyzer ............................................................................................ 5
   2.4 Grail .......................................................................................................... 6
   2.5 yEd ........................................................................................................... 6
3. State of the Art............................................................................................... 7
   3.1 Candidates................................................................................................. 7
      3.1.1 Eli .................................................................................................... 7
      3.1.2 Open 64 ........................................................................................ 7
      3.1.3 Eclipse CDT..................................................................................... 7
   3.2 Discussion:................................................................................................ 8
      3.2.1 Criteria............................................................................................. 8
      3.2.2 Mapping table .................................................................................. 8
4. Requirements ................................................................................................. 9
   4.1 Overall Description ................................................................................... 9
      4.1.1 Product perspective .......................................................................... 9
   4.2 Functional Requirements ........................................................................... 9
   4.3 Non-functional Requirements .....................................................................11
5. Architecture ..................................................................................................12
   5.1 C/C++ Meta-Model Architecture..............................................................12
   5.2 Package Structure.....................................................................................13
   5.3 Package Interaction...................................................................................15
   5.4 Class Diagrams ........................................................................................17
      5.4.1 In package “core”............................................................................17
      5.4.2 In package “eclipsePlugin” ..............................................................22
6. Design ..........................................................................................................26
   6.1 Meta-Model Mapping...............................................................................26
      6.1.1 Common-Meta-Model ..................................................................26
      6.1.2 C/C++ Specific Meta-Model ................................................................27
      6.1.3 Mapping CDT ASTNode to C/C++ Model Element .......................29
   6.2 Node Key Generating...............................................................................32
      6.2.1 Key Generating Algorithm in JDT...................................................32
      6.2.2 Key Generating Algorithm in C/C++................................................33
      6.2.3 Example of Key Generating Algorithm in C++.................................33
   6.3 Steps of Creating Node and Edge .............................................................34
7. Implementation.............................................................................................35
   7.1 Start-up procedure ...................................................................................35
      7.1.1 Get and install Eclipse CDT: ...........................................................35
      7.1.2 Create a new Plug-in project: ..........................................................35
   7.2 Get Source Files .......................................................................................36
   7.3 Generate Key ..........................................................................................36
7.4 Create a Node .......................................................................................... 37
7.5 Create a Edge .......................................................................................... 38
8. Conclusion and Future Work ...................................................................... 42
  8.1 Conclusions ............................................................................................ 42
  8.2 Future work ............................................................................................ 43
Reference ........................................................................................................... 44
Figures

Figure 2-1: Program Analysis Process ................................................................. 3
Figure 2-2: Meta-Model Theory ........................................................................ 4
Figure 2-3: A front-end specific, a common, and a view model ....................... 5
Figure 2-4: VizzAnalyzer Framework ............................................................... 6
Figure 5-1: C/C++ Meta-Model Architecture ................................................... 13
Figure 5-2: Package Structure ......................................................................... 14
Figure 5-3: Package Interaction ....................................................................... 16
Figure 5-4: Classes in package "core" .............................................................. 17
Figure 5-5: Classes in package "eclipsePlugin" ................................................ 22
Figure 6-1: Example of constructor ............................................................... 31
Figure 6-2: Generating key Example Program -- main.cpp ............................ 33
Figure 6-3: Generating key Example Program -- ExampleClass.cpp ............. 33
Figure 6-4: Generating key Example Program -- ExampleClass.h ................. 34
Tables

Table 3-1: Detailed description for Eli ................................................................. 7
Table 3-2: Detailed description for Open 64.......................................................... 7
Table 3-3: Detailed description for Eclipse CDT................................................... 8
Table 3-4: Criteria for State of the art................................................................. 8
Table 3-5: Mapping between Candidates and Criteria .......................................... 8
Table 6-1: CMM, productions and specialization hierarchy.................................... 27
Table 6-2: Binary semantic relations in Common-meta-model ................................ 27
Table 6-3: Specialization hierarchy of semantic relations ..................................... 27
Table 6-4: Productions and Specialization hierarchy in C .................................... 28
Table 6-5: Binary semantic Relations in C ........................................................... 28
Table 6-6: Mapping Function $\alpha_{cpp}$ between C Meta-Model and CMM ............. 28
Table 6-7: Productions and Specialization hierarchy in C++ ................................ 29
Table 6-8: Binary semantic Relations in C++ ...................................................... 29
Table 6-9: Mapping Function $\alpha_{cpp}$ between C++ Meta-Model and CMM ......... 29
Table 6-10: Mapping between C Meta-Model and CDT ASTNodes ..................... 29
Table 6-11: Mapping between C++ Meta-Model and CDT ASTNodes ................. 30
Table 6-12: Key Generating Algorithm in JDT [1] ............................................. 32
Table 6-13: Key Generating Algorithm in C/C++ .............................................. 33
Table 6-14: Example key generating result....................................................... 34
1. Introduction

Relevant studies about Software maintenance show that cost on maintenance work is enormous. Some evidence even prove that the relative cost for maintaining software and managing its evolution now represents more than 90\%.[1] These facts indicate the importance of doing program analysis which helps developers to understand software systems better and provide an access to software quality.

Besides, Lincke el. al. is currently engaged in validating a standard-and metric-based software quality model for program analysis.[2] To achieve this goal, one of the problems which need to be solved is to make the quality model programming language-independent. The mechanism applied for doing this is to use a Common-Meta-Model according to Lincke’s work. Based on the Common-Meta-Model, we can create front-end Meta-Model for any program language. As proof of this concept, his work has been implemented in the VizzAnalyzer, which are Java specific Front-End and UML Front-End. Using these two Front-Ends, Java source code and UML can be analyzed by VizzAnalyzer currently.

Considering C and C++ are two program languages with high popularity nowadays,[3] we want to be able to analyze C/C++ programs as well; therefore a new front-end allowing retrieving information from C/C++ source code needs to be constructed.

1.1 Problem

The problem of this thesis can be summarized as:

*The problem tackled by this thesis is to define C/C++ front-end allowing analyzing C/C++ program in the VizzAnalyzer.*

This is a complex problem because it requires understanding C and C++ which are two complex programming languages to define Front-End specific Meta-Model for them and to create mapping between C/C++ Meta-Model and Common-Meta-Model. Additionally, in order to get C/C++ Meta-Model, a compiler/parser shall be used for C/C++ programs. Therefore, the compiler/parser should be understood as well.

Common-Meta-Model and C/C++ language specification will be used as two major inputs to decide which nodes and edges need to be created. Besides these two, the requirements from analysis part of VizzAnalyzer will also be taken into concern to help us decide which nodes and edges are the most important.

1.2 Goals and Criteria

This section describes the goals pursued by this thesis in order to solve the problem and the criteria used for validating the goal:

- Our first goal is to define C/C++ front-end Meta-Model: We should define a C/C++ front-end meta-model according to C/C++ language specification. Because the Meta-model will be used for metrics analyzing, it is not necessary to include language specification of C/C++ entirely into our Meta-model. Only information needed by certain metrics will be taken into concern at present. The goal is reached when C/C++ program has been parsed by a certain compiler.
- The second goal is to define mapping for Common-meta-model: Based on Meta-model Theory, this mapping step is crucial for this C/C++ front-end. We define a set of mapping functions to ensure the “safe” transformation between C/C++ specific Meta-model and Common-Meta-Model. This goal
is reached when C/C++ front-end Meta-Model has been traversed, useful information has been gathered and a new AST conformed to Common-Meta-Model has been created.

- The third goal is to use suitable infrastructure: The C/C++ front-end is build for VizzAnalyzer. VizzAnalyzer is mainly working on Eclipse, so if it is possible, we should use Eclipse as platform.
- The last goal is to use event-based architecture: the C/C++ front-end should reuse event-based architecture defined by Lincke el. al. for Retrieval Plugin of VizzAnalyzer.

The outcome of this thesis is a plug-in for VizzAnalyzer. VizzAnalyzer will use the information got from C/C++ front-end plug-in in its further analysis.

1.3 Motivation
C and C++ are two of the most widely accepted programming languages nowadays. However, our VizzAnalyzer currently only supports analyzing Java programs and UML while C and C++ are still out of its scope. We want to extend our research to involve C/C++ programs in order to make VizzAnalyzer available in larger area and to be accepted by more users.

Based on this, a C/C++ front-end which defined mapping between C/C++ front-end meta-model and Common-meta-model is desired by us.

1.4 Outline
The structure of this thesis is as follows. Chapter 2 gives some background knowledge, which is essential to understand the C/C++ Front-End. The background covers several theoretical aspects of C/C++ Front-End and explains some concepts and tools mentioned often in this thesis. Chapter 3 describes the tools, which can be used to create C/C++ front-end. We did some research into several candidate tools and made a decision which one to use. Chapter 4 is for requirements. We describe functional requirements and non-functional requirements of this project. Chapter 5 introduces architecture of C/C++ front-end. The architecture explains the position of C/C++ front-end in VizzAnalyzer and the internal organization of C/C++ front-end. Chapter 6 covers several design aspects from mapping between Common-Meta-Model to C/C++ specific Meta-Model, nodes and edges defined in C/C++ front-end, node key generating algorithm and so on. Chapter 7 focuses on implementation aspect, describes implementation detail of setting working environment (Eclipse plug-in) and creating nodes and edges. Chapter 8 makes a conclusion of the thesis and describes future work.
2. Background
This section provides some background needed to understand the thesis. We start from the theoretical issues, the concept of program analysis process. In this process, we are particularly interested in the Software Analysis part. Therefore, we discuss it in detail in section 2.2 Meta-model theory. Secondly, we introduce the VizzAnalyzer Framework which is constructed according to program analysis process and meta-model theory. In the end, Grail and yEd are described separated in a brief way.

2.1 Program Analysis Process
Figure 2-1 shows the whole process of software analysis and software visualization. [4]
There are two steps to be done before the launch of software visualization: information extraction and information analysis. For the first step, we can take advantage of parsers. After parsing, an Abstract Syntax Tree (AST) will be generated which keeps the original information of source documents. In other words, this AST is another representation of source documents. This can be regarded as ‘Base Model’ here. Then, a further step needs to be done since the parsing information cannot be used directly by software visualization. Information tailoring is necessary here which means we need to work on the transformation from Base Model to Model.

What a Front-end does is to accomplish part of the transformation from Base Model to Model. Firstly, it gathers information from Base Model and then after some information processing, it creates an intermediate model (between Base Model to Model) which will be used in further analysis.

So far, we already have front-end for Java and UML, but still lack the one for C/C++. The thesis is aimed at creating the front-end for C/C++.

Since the thesis is particularly related to the process from Source Documents to Model, we will explain it in next section Meta-Model Theory in more detail.

![Figure 2-1: Program Analysis Process](image-url)
2.2 Meta-Model Theory

Figure 2-2 describes the architecture of Meta-model theory. Meta-model theory is based on a three-layer architecture. From top to down, Meta-meta-model to Model, each layer is less abstract than its previous one. Meta-meta Model layer is the most abstract one which specifies the basic representation of the other two models. Here, tree grammars and relations are being selected. Then, the second layer, Meta-model is less related to concrete programs, in other words, it is an abstract expression of Model which lies on the bottom.

Model and Meta-model are described according to information processing from source documents to metrics analysis which is divided into four steps: Information extraction, mapping to common representation, view abstraction and analysis.

The beginning step is information extraction which takes source files as input. After parsing the source file (information extraction), a special model reflecting the original structure of source file will be created. This step is described in Figure 2-1 as transformation from source file to base model. Each front-end meta-model gives a specific program representation. Since result of this step is completely depending on source file, we cannot influence it.

The second step is language mapping. A language mapping is a set of defined functions specifying how elements belonging to one meta-model are mapped to elements of another meta-model. Particularly, in this thesis, language mapping exists between C/C++ front-end specific meta-model and
Common-Meta-Model. Common-Meta-Model is an abstraction from front-end specific details. It is designed to ensure analyses are language-independent. Because Common-meta-model will be used by analysis, it should always evolve according to the requirements from analyses.

Next two steps are view abstraction and analysis. View abstraction provides an intermediate between common-meta-model and analysis which make it possible for analysis operated on some certain views instead of directly on common-meta-model.

Figure 2-3 gives us an example to show how front-end specific model, common model and view model work. The first diagram contains more information than the other two because it reflects the original structure of source file and hasn’t been tailored. In the second diagram, it can be seen that several nodes and edges have been discarded from diagram 1 to adapt the front-end specific model into common model. The last one is a view model which selects some nodes and edges from common-model according to the requirements of certain metrics.

Task of this thesis is to create the mapping between C/C++ front-end specific model and Common-meta-model. This is the second step of information processing explained above. It will take AST generated by Eclipse CDT as input and creates a new tree according to Common-meta-model.

2.3 VizzAnalyzer

VizzAnalyzer is a stand-alone tool developed in Växjö University based on the software analysis process and Common-meta-model theory which have been explained in the two previous sections. Three sub-processes “Retrieval”, “Analysis” and “Visualization” exist. “Retrieval” can also be called as “Front-end” which provides prerequisites for further analysis. It covers the whole “information extraction” and part of “analysis” in Figure 2.1. “Analysis” in VizzAnalyzer is corresponding to the rest part of “analysis” in Figure 2.1. And the last sub-process “Visualization” focuses exactly on “Information Visualization” Domain.
VizzAnalyzer has a flexible architecture that each of the sub-processes is defined by reverse engineering components, which are plugged via wrappers into respective extension points. In Figure 2-4, “Front-end”, “Analyzer”, “Metrics”, “yEd”, “Vizz3d” all play a plug-in role in VizzAnalyzer. [7]

“Grail” in the Figure is an internal data representation of VizzAnalyzer. Further information can be found in the next section Grail.

C/C++ front-end is a plug-in for “Retrieval” sub-process, which will be used to analysis C/C++ source code. VizzAnalyzer will take result of C/C++ front-end to do further analysis.

2.4 Grail
GRAIL is the abbreviation of GRAph ImpLementation. It is a graph library created by development team of VizzAnalyzer, which is regarded as an internal data representation for VizzAnalyzer. In particularly, common-meta-model is stored using data structure of Grail.

This representation consists of an annotated graph, where each graph entity (nodes, edges, and the graph itself) has a data object and a set of predicates attached to it. [8]

The mainly usage of Grail in this project is to provide node and edge type definitions (grail.properties.typeValues). Besides, in the end, method in grail needs to be called to generate a GML file for the tree generated by C/C++ front-end.

2.5 yEd
yEd is a graph editor which is completely written in Java. yEd supports generating graphs and especially, it can be used to lay out complex graph structures in an efficient and automatic manner. Several different kinds of layout are available at present, each of which has its own features and advantages. [9]

Grail can export graph into gml file (graph markup language) which can be displayed by yEd. Thus, graph of Grail can be easily viewed for debugging and other purposes.

yEd can be found on http://www.yworks.com/en/products_yed_about.htm, and is available to be downloaded for free.
3. State of the Art

This section describes several tools which could be used to parse C/C++ programs and to provide suitable Abstract Syntax Tree for further information extraction and tree-traversing mechanism. First, we give three candidate tools with some detail description. Secondly, several criteria for selecting the suitable tool are raised and a mapping between criteria and candidates is created. In the last section, we give our selection and reasons for selecting that tool.

3.1 Candidates

Here are three candidate tools. Each one has a name, vendor, website and brief description.

3.1.1 Eli

<table>
<thead>
<tr>
<th>Name:</th>
<th>Eli</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor:</td>
<td>Non-commercial tool</td>
</tr>
<tr>
<td>Website:</td>
<td><a href="http://ag-kastens.uni-paderborn.de/eli_homeE.html">http://ag-kastens.uni-paderborn.de/eli_homeE.html</a></td>
</tr>
</tbody>
</table>
| Description | Eli is a domain-specific programming environment combining some standard tools that implement compiler construction strategies. It has programming language compilers for
  - ANSI C to SPARC machine code
  - Pascal to Pcode
  - Pascal to C
  - Some other user-defined languages [10] |

Table 3-1: Detailed description for Eli

3.1.2 Open 64

<table>
<thead>
<tr>
<th>Name:</th>
<th>Open 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor:</td>
<td>Non-commercial tool</td>
</tr>
<tr>
<td>Website:</td>
<td><a href="http://www.open64.net">http://www.open64.net</a></td>
</tr>
<tr>
<td>Description</td>
<td>The Open64 compiler suite currently includes compilers for C, C++, and Fortran90/95 compilers for the IA-64 Linux ABI and API standards. Open64 is currently in version 3.1. [11]</td>
</tr>
</tbody>
</table>

Table 3-2: Detailed description for Open 64

3.1.3 Eclipse CDT

<table>
<thead>
<tr>
<th>Name:</th>
<th>Eclipse CDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor:</td>
<td>Non-commercial tool</td>
</tr>
<tr>
<td>Website:</td>
<td><a href="http://www.eclipse.org/cdt/">http://www.eclipse.org/cdt/</a></td>
</tr>
<tr>
<td>Description</td>
<td>CDT is shortening form of “C/C++ Development Tools”. It adds C/C++ project structure and syntax recognition to the Eclipse platform. CDT does not include its own set of build tools, relying instead on the GNU tool chain (by default). It is an industrial strength C/C++ IDE that also serves as a</td>
</tr>
</tbody>
</table>
platform for others to provide value added tooling for C/C++ developers. [12]

Table 3-3: Detailed description for Eclipse CDT

3.2 Discussion:
To decide which tool to use, we need to give selection criteria and to have a look at whether the tools meet our criteria.

3.2.1 Criteria

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAC1</td>
<td>The tool should be non-commercial.</td>
</tr>
<tr>
<td>SAC2</td>
<td>The tool should be easy to be installed and used on a variety of platforms, such as Windows, Linux, Mac OS X.</td>
</tr>
<tr>
<td>SAC3</td>
<td>The tool should have sufficient information related to it which helps it to be understood in an acceptable time frame.</td>
</tr>
<tr>
<td>SAC4</td>
<td>Since C/C++ front-end is a plug-in for VizzAnalyzer, the tool should be suitable for VizzAnalyzer to use.</td>
</tr>
<tr>
<td>SAC5</td>
<td>The tool should be up-to-date and have a long-term perspective.</td>
</tr>
</tbody>
</table>

Table 3-4: Criteria for State of the art

3.2.2 Mapping table

<table>
<thead>
<tr>
<th>Candidate/ Criteria</th>
<th>SAC1</th>
<th>SAC2</th>
<th>SAC2</th>
<th>SAC4</th>
<th>SAC5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eli:</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Unknown</td>
</tr>
<tr>
<td>Open 64:</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Unknown</td>
</tr>
<tr>
<td>Eclipse CDT:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3-5: Mapping between Candidates and Criteria

From the mapping table, we selected Eclipse CDT as the tool used to develop the C/C++ front-end. The reasons can be stated as following again:

Eclipse is a worldwide used open development platform which supports a variety of popular programming languages, including Java, C/C++, PHP and so on. Besides, it is a non-commercial tool under healthy development. It can be easily installed and used on almost all platforms exist nowadays. The Eclipse Community is active which will also provide some information for the development of C/C++ front-end. Another concern is our VizzAnalyzer is recently been updated to version 2.0 which works as plug-in of Eclipse. Considering about the reasons above, we think Eclipse CDT is a proper alternative for us to develop the C/C++ front-end.
4. Requirements
This chapter describes requirements of C/C++ Front-end. Requirements are described in three sections. The first section gives an overview of C/C++ front-end, the second one is functional-requirement part and the last one covers all the non-functional requirements.

4.1 Overall Description
C/C++ front-end is a program will be used by VizzAnalyzer to analyze C/C++ programs. This section gives detailed description about product perspective to help understand it from several aspects.

4.1.1 Product perspective
User Interfaces:
C/C++ Front-end will be used by VizzAnalyzer, so there is no user interface required.

Hardware Interfaces:
C/C++ Front-end doesn’t need to interactive with hardware. We don’t need to take care of hardware interfaces.

Software Interfaces:
C/C++ front-end uses Eclipse CDT as the compiler of C/C++ programs and is built as a plug-in of Eclipse. Thus, there is interface for interacting with Eclipse. Additionally, C/C++ front-end will be used by VizzAnalyzer, interface of VizzAnalyzer is needed.

Memory Constraints:
We want C/C++ front-end is able to run fast. But most time will spend on parsing and traversing programs which largely depends on the size of programs. Therefore, if the programs need to analyzed are in large scale, more memory will be required.

4.2 Functional Requirements
In this section, we list the function requirements of C/C++ front-end, focusing on what should be done by C/C++ front-end. Each of the functional requirements has a brief description, pro-condition, post-condition, related FR and priority.

<table>
<thead>
<tr>
<th>FR1:</th>
<th>Get source files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>It should be able to get all source files from open projects in a WorkBenchWindow.</td>
</tr>
<tr>
<td>Pre-condition:</td>
<td>There are some C/C++ source files in currently opened WorkBenchWindow.</td>
</tr>
<tr>
<td>Post-condition:</td>
<td>Source file has been recognized, read and prepared to be traversed.</td>
</tr>
<tr>
<td>Related FR:</td>
<td>FR2: Traverse source files</td>
</tr>
<tr>
<td>Priority:</td>
<td>Essential</td>
</tr>
</tbody>
</table>

FR2: Traverse source files
<table>
<thead>
<tr>
<th>Description</th>
<th>It should be have mechanism to traverse source files.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Post-condition:</strong></td>
<td>Source file has been traversed, useful information has been gathered, process of mapping C/C++ front-end Meta-Model to Common-Meta-Model has been finished, and a new AST has been constructed.</td>
</tr>
<tr>
<td><strong>Related FR:</strong></td>
<td>Contains FR2.1 Get information from source files Contains FR2.2 Create a new AST</td>
</tr>
<tr>
<td><strong>Priority:</strong></td>
<td>Essential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>FR2.1:</strong></th>
<th>Get information from source files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>When it is traversing source file, it should be able to get information needed to map C/C++ front-end meta-model to Common-Meta-Model.</td>
</tr>
<tr>
<td><strong>Post-condition:</strong></td>
<td>A source file is under traversing.</td>
</tr>
<tr>
<td><strong>Related FR:</strong></td>
<td>FR2: Traverse source files FR2.2: Create a new AST</td>
</tr>
<tr>
<td><strong>Priority:</strong></td>
<td>Essential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>FR2.2:</strong></th>
<th>Create a new AST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>Create a AST using input information.</td>
</tr>
<tr>
<td><strong>Post-condition:</strong></td>
<td>FR2.1 has been processed.</td>
</tr>
<tr>
<td><strong>Related FR:</strong></td>
<td>FR2.1: Get information from source files FR2.3: Create a graph containing a AST</td>
</tr>
<tr>
<td><strong>Priority:</strong></td>
<td>Essential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>FR2.3:</strong></th>
<th>Create a graph containing a AST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>Create a graph using information of a AST, get from FR2.2</td>
</tr>
<tr>
<td><strong>Post-condition:</strong></td>
<td>FR2.2 has been processed.</td>
</tr>
<tr>
<td><strong>Related FR:</strong></td>
<td>FR2.2: Get information from source files FR3: Generate gml file</td>
</tr>
<tr>
<td><strong>Priority:</strong></td>
<td>Desirable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>FR3:</strong></th>
<th>Generate gml file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>It shall be able to generate a gml file.</td>
</tr>
<tr>
<td><strong>Post-condition:</strong></td>
<td>Graph got from FR2.3 is converted into a gml file.</td>
</tr>
</tbody>
</table>
4.3 Non-functional Requirements

<table>
<thead>
<tr>
<th>NFR1:</th>
<th>Based on existing theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Its design should be based on existing theory, such Meta-model Theory, event-based architecture.</td>
</tr>
<tr>
<td>Priority:</td>
<td>Essential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NFR2:</th>
<th>Suitable for VizzAnalyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>It should be suitable for VizzAnalyzer to use.</td>
</tr>
<tr>
<td>Priority:</td>
<td>Essential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NFR3:</th>
<th>Result should be repeatable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Result got from the system should be repeatable.</td>
</tr>
<tr>
<td>Priority:</td>
<td>Essential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NFR4:</th>
<th>Design in a clear way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Its design and implementation should be easily understood.</td>
</tr>
<tr>
<td>Priority:</td>
<td>Desirable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NFR5:</th>
<th>Run fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>It should be fast enough (related to memory constraint)</td>
</tr>
<tr>
<td>Priority:</td>
<td>Desirable</td>
</tr>
</tbody>
</table>
5. Architecture

This chapter describes the architecture of C/C++ Front-end. The first section explains how C/C++ front-end adapts itself to Meta-model theory and how event-based architecture is used. Next two sections are mainly related to package architecture, the first of them is package structure which the other one focuses on interaction between packages. The last part of this chapter is class diagram which gives a detail view how the classes in C/C++ front-end work.

5.1 C/C++ Meta-Model Architecture

Figure 5-1 shows how the C/C++ front-end conforms to Meta-Model theory which is explained in section 2.2. Besides, this figure also describes how event-based architecture is applied in C/C++ front-end.

C/C++ front-end covers first two of the total four steps in Meta-Model Theory: Information Extraction and Mapping to language-independent representation.

In the first step, we take advantage of Eclipse CDT. After parsing by CDT, C/C++ source file is interpreted into a AST structure with predefined ASTNodes and “contains” relationship between nodes.

The second step reuses an Event-based architecture which is defined by Lincke el. al. [13] and has been introduced to Java and UML front-end. As depicted in Figure 5-1, language mapping step is divided into two parts, MapsA and MapsB in the figure respectively. In MapsA, a “visitor” traverses the original AST created by Eclipse CDT and sends events to a Backend Listener which gathers information from MapsA and constructs a new AST which is corresponding to Common-meta model.

To understand the architecture better, we need to explain Backend listener briefly first. Backend is a simple and flexible interface which actually can connect with several “traversing visitor” in MapsA. The major usage of it is to create a tree with nodes and reference relations between nodes. Five methods exist in Backend, their names are:

- addLeaf
- enterNode
- exitNode
- addCrossReferenceEdge
- completeGraphs

The methods responsibilities are intuitive to be understood from their names.

A back-end has already existed for Java and UML front-end. We can reuse it directly in the C/C++ front-end.

The major task of this thesis is to create a C/C++ specific traversing visitor which will take AST from Eclipse CDT as input and cooperate with the Backend.
5.2 Package Structure

C/C++ front-end is created as a plug-in for VizzAnalyzer which will be integrated into VizzAnalyzer. Figure 5-2 depicts the position of CandCpp front-end in VizzAnalyzer and the internal package structure of CandCpp.

From VizzAnalyzer point of view, CandCpp is a sub-package of package “frontEnds”. There could be more front-ends for other program language inside package “frontEnds”.

Inside CandCpp, there are four packages, three of them c, cpp and util are responsible for the major part of work, such as traversing the source file AST, extracting useful information. Since this project is a Eclipse plug-in, another package “eclipsePlugin” mainly deals with Eclipse plug-in affairs.

Here is the description for each package:

<table>
<thead>
<tr>
<th>Package 1: core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
</tr>
</tbody>
</table>
| **Corresponding classes:** | 1. EclipseCFrontendNode  
2. EclipseCppFrontendNode  
3. Statistics  
4. Version |
| **Relations:** | is sub-package of CandCpp package used by eclipsePlugin package |
| **Sub packages:** | package 1.1 c, package 1.2 cpp, package 1.3 util |
### Package 1.1: c

**Description:** This package is designed for C programs. It is major part in core package; responsibility is similar to the responsible of package core.

**Corresponding classes:**
1. EclipseCFrontendNode

**Relations:**
is sub-package of package 1 core
use package 1.3 util

**Sub packages:** None.

### Package 1.2: cpp

**Description:** This package is designed for C++ programs. It works in similar way as package 1.1 c

**Corresponding classes:**
1. EclipseCppFrontendNode

**Relations:**
is sub-package of package 1 core
use package 1.3 util

**Sub packages:** None.

### Package 1.3: util

**Description:** This package is assistance package for package 1.1 c and package 1.2 cpp. Package c and cpp take this package as a repository to store some value temporarily.

**Corresponding classes:**
1. Statistics

**Relations:**
is sub-package of package 1 core
use package 1.3 util

**Sub packages:** None.

### Package 2: eclipsePlugin

**Description:** This package is used to do the routine work as an Eclipse plug-in.
Besides, it communicates with package 1 core, providing information to it.

### Corresponding classes:
1. Activator
2. Action
3. GenerateGML
4. UserInteraction

### Relations:
is sub-package of package 1 core
use package 1.3 util

### Sub packages:
package 2.1 actions

### Package 2.1: actions

<table>
<thead>
<tr>
<th>Description:</th>
<th>This package is responsible to give response to actions in plug-in projects.</th>
</tr>
</thead>
</table>
| Corresponding classes: | 1. Action  
2. GenerateGML  
3. UserInteraction |
| Relations: | is sub-package of package 2 eclipsePlugin |
| Sub packages: | None. |

### 5.3 Package Interaction

C/C++ front-end needs to cooperate with other components in VizzAnalyzer. Figure 5.3 shows these external jar files and how CandCpp communicate with them.

### External Package 1: attribute

<table>
<thead>
<tr>
<th>Description:</th>
<th>This package is classes which defines graph attributes.</th>
</tr>
</thead>
</table>
| Corresponding classes: | Attribute  
Attributes |
| Relationship with “CandCpp” | This package provides CandCpp access to “graph attributes” which will be used when adding a new node and edge into graph. |

### External Package 2: model

<table>
<thead>
<tr>
<th>Description:</th>
<th>This package contains only one class – CommonMetaModel10, which provides definition of Common-meta-model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corresponding classes:</td>
<td>CommonMetaModel10</td>
</tr>
<tr>
<td>Relationship with “CandCpp”</td>
<td>The task of C/C++ front-end is to create mapping between C/C++ specific front-end model and Common-meta-model. C/C++ front-end model will be mapped to Common-meta-model.</td>
</tr>
</tbody>
</table>
Figure 5-3: Package Interaction

### External Package 1: backend

**Description:** This package is actually part of “front-end”, it provides methods to construct a tree. (see also Chapter 2.5 Relationship with “CandCpp”)

**Corresponding classes:**
- DefaultBackEnd
- VizzBackEnd

**Relationship with “CandCpp”**
Backend works together with “source code reader”, now actually the CandCpp part to accomplish building a new tree.

### External Package 4: grail

**Description:** This package is an internal package of VizzAnalyzer which works as a graph data representation. (see also Chapter 2.6 Grail)

**Relationship with “CandCpp”**
It mainly provides type definition to package eclipsePlugin and core. Besides, after traversing the AST and accomplishing building the new AST, a gml file which contains the new AST information needs to be generated, GenerateGML method in Grail will be called and some other declarations and methods related to graph will be used.
5.4 Class Diagrams
This section gives some description of classes. We organize the classes in the unit of package. Each class has information about name, in which package, responsibility, relation with other classes, methods description and attributes description.

5.4.1 In package “core”
Figure 5-4 shows the hierarchy of classes in package “core”. Firstly, classes on the top: ASTVisitor, CASTVisitor, CppASTVisitor are three abstract classes provided by CDT which are used to traverse AST data structure. EclipseCFrontEnd and EclipseCppFrontEnd extend from CASTVisitor and CppASTVisitor respectively. These two extended classes play “traversing visitor” role which is described in Figure 5-1. Statistics class works as an assistant class stores some information which will be used later by EclipseCFrontEnd and EclipseCppFrontEnd. The last one, Version is class simply used to store and provide version information of C/C++ front-end which is less dependent on others.
### Version

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Version</td>
</tr>
<tr>
<td>Package in</td>
<td>package 1: core</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Provides version information</td>
</tr>
<tr>
<td>Relations</td>
<td>None</td>
</tr>
<tr>
<td>Methods</td>
<td>1. getVersion: return version information</td>
</tr>
<tr>
<td>Attributes</td>
<td>1. version: stores version information</td>
</tr>
</tbody>
</table>

### EclipseCFrontEnd

```java
class EclipseCFrontEnd {
    -listener: VizzBackEnd
    -debug: bool
    final static
    -file: String
    -functionKeyStack: Stack<String>
    -variableKeyStack: Stack<String>
    -stats: Statistics
    -parameterTypes: String
    -CONSOLE: String = VizzAnalyzer
    static final
    -console: MessageConsoleStream = findConsole(CONSOLE).newMessageStream
    final

    +EclipseCFrontEnd(listener: VizzBackEnd, String: fileName, String: folderName)
    +visit(tu: IASTTranslationUnit): int
    +visit(in node: IASTNode): int
    +leave(tu: IASTTranslationUnit)
    int
    +leave(node: IASTNode)
    -findConsole(name: String): MessageConsole
    -getParameterTypes(expression: IASTExpression)
}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>EclipseCFrontEnd</td>
</tr>
<tr>
<td>Package in</td>
<td>package 1.1 c</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Traverse a C AST tree, gather useful information during traversing and send information to backend (an external component).</td>
</tr>
<tr>
<td>Relations</td>
<td>Extends CASTVisitor which is provided by CDT</td>
</tr>
<tr>
<td>Uses Statistics</td>
<td></td>
</tr>
</tbody>
</table>

---

18
| Methods | 1. EclipseCFrontEnd: constructor, initial class fields, set all boolean values like “shouldVisitNames” as true which means we are going to visit all kinds of ASTNodes.  
2. visit: for each kind of ASTNode, there is a “visit” method. This method is used to visit ASTNode, during the visit, useful information will be gathered. Working together with backend, new nodes and edges will be created.  
3. leave: for each kind of ASTNode, there is a “leave” method. When comes to this method, it means a certain node has been visited.  
4. findConsole: find current console, if there is no console available, a new one will be created.  
5. getParameterTypes: used to get function parameter types. This information is needed when a function key is needed. |
| Attributes | 1. listener: an VizzBackEnd instance  
2. debug: whether it is in debug mode. If it is in debug mode, more information will be stored into the new tree created by backend  
3. file: the current file name  
4. variableKeyStack: stack for variableKey. Works in similar as classKeyStack.  
5. stats: an instance of Statistics.  
6. parameterTypes: a string contains all the parameters types names of a function  
7. CONSOLE: console name is “VizzAnalyzer”  
8. console: an console instance |
### EclipseCppFrontEnd

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>EclipseCppFrontEnd</td>
</tr>
<tr>
<td>Package in</td>
<td>package 1.2 cpp</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Traverse a CPP AST tree, gather useful information during traversing and send information to backend (an external component). Uses Statistics</td>
</tr>
<tr>
<td>Relations</td>
<td>Extends CPPASTVisitor which is provided by CDT</td>
</tr>
</tbody>
</table>
| Methods        | 1. EclipseCppFrontEnd: constructor, initial class fields, set all boolean values like “shouldVisitNames” as true which means we are going to visit all kinds of ASTNodes.  
2. visit: for each kind of ASTNode, there is a “visit” method. This method is used to visit ASTNode, during the visit, useful information will be gathered. Working together with backend, new nodes and edges will be created.  
3. leave: for each kind of ASTNode, there is a “leave” method. When comes to this method, it means a certain node has been visited.  
4. findConsole: find current console, if there is no console available, a new one will be created.  
5. getParameterTypes: used to get function parameter types. This information is needed when a function key is needed. |
| Attributes      | 1. listener: an VizzBackEnd instance  
2. debug: whether it is in debug mode. If it is in debug mode, |
more information will be stored into the new tree created by backend
3. file: the current file name
4. classKeyStack: stack for classKey. A classKey will be pushed into stack when visiting a class node, will be popped when leaving a class node.
5. fieldKeyStack: stack for fieldKey. Works in similar as classKeyStack.
6. variableKeyStack: stack for variableKey. Works in similar as classKeyStack.
7. constructorKeyStack: stack for constructorKey. Works in similar as classKeyStack.
8. stats: an instance of Statistics.
9. parameterTypes:
10. CONSOLE: console name is “VizzAnalyzer”
11. console: an console instance

Statistics

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Statistics</td>
</tr>
<tr>
<td>Package in</td>
<td>package 1.3 util</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Store some information temporally for EclipseCFrontEnd or EclipseCppFrontEnd, and provide methods to operate on those information. Most of the information are used to generate keys, some of the information are used to locate IASTField nodes.</td>
</tr>
<tr>
<td>Relations</td>
<td>Used by EclipseCFrontEnd &amp; EclipseCppFrontEnd</td>
</tr>
<tr>
<td>Methods</td>
<td>1. getInstance: get a instance of this class</td>
</tr>
<tr>
<td></td>
<td>2. addASTQualifiedName: when find a ASTQualifiedName node during traversing source file, add</td>
</tr>
</tbody>
</table>
3. getFunctionASTName: go through Vector qualifiedName, to see whether one of the qualified names has the name equal to parameter “functionName”
4. addClassASTName: when find a class node during traversing source file, add the name into Vector <classNames>
5. addField: when find a field node, add into Vector <fields>
6. getFunctionName: using a name as parameter, go through Vector qualifiedNames, to see whether there is a function node corresponding to the parameter “name”.
7. findConsole: find current console, if there is no console available, create a new one.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. stats: an instance of statistics</td>
<td></td>
</tr>
<tr>
<td>2. qualifiedNames: Vector used to store qualified name nodes</td>
<td></td>
</tr>
<tr>
<td>3. fields: Vector used to store field nodes</td>
<td></td>
</tr>
<tr>
<td>4. classNames: Vector used to store class names</td>
<td></td>
</tr>
<tr>
<td>5. CONSOLE: name of the console</td>
<td></td>
</tr>
<tr>
<td>6. console: an console instance</td>
<td></td>
</tr>
</tbody>
</table>

**5.4.2 In package “eclipsePlugin”**

Figure 5-5 shows class hierarchy in package eclipsePlugin. This is a package used to infrastructure work as an Eclipse Plug-in (class Activator and Action). Since this package generally focuses on the first step “information extraction” described in Figure 5-1. It has class GenerateGML to provide input to classes in package “core” for further work. Another task of this package is to interaction with user; the class UserInteraction designed for the purpose.

![Figure 5-5: Classes in package "eclipsePlugin"](image-url)
**Activator**

```java
package 2 eclipsePlugin

Activator
    extends AbstractUIPlugin

+PLUGIN_ID: String = C_CPPFrontEnd
static final

-plugin: Activator
static

+start(context:BundleContext)
+stop(context:BundleContext)
-getDefault(): Activator
static
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Activator</td>
</tr>
<tr>
<td>Package in</td>
<td>package 2 eclipsePlugin</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>used to control a plug-in project</td>
</tr>
<tr>
<td>Relations</td>
<td>extends from AbstractUIPlugin</td>
</tr>
</tbody>
</table>

**Methods**

1. start: start a plug-in project
2. stop: stop a plug-in project
3. getDefault: get the shared instance of this plug-in project.

**Attributes**

1. PLUGIN_ID: id of this plug-in project
2. plugin: a shared instance of this project

---

**Action**

```java
package 2.1 Actions

Action
    implements IWorkbenchWindow
    ActionDelegate

-log: Logger = Logger.getLogger("plug-in")
-logWindow: IWorkbenchWindow
-uniqInteraction: UserInteraction
-generateGML: GenerateGML
-CONSOLE: String = VizzAnalyzer
static final

-console: MessageConsoleStream = findConsole(CONSOLE).newMessageStream
static final

+Action()
+dispose()
+init(in window:IWorkbenchWindow)
+selectionChanged(action:IAction, selection:ISelection)
+run(action:IAction)
+getWindow(): IWorkbenchWindow
+findConsole(name:String): MessageConsole
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Action</td>
</tr>
<tr>
<td>Package in</td>
<td>package 2.1 Actions</td>
</tr>
</tbody>
</table>
### Responsibilities

Used to deal with actions from a plug-in project.

### Relations

Implements IWorkbenchWindowActionDelegate
Uses class GenerateGML and UserInteraction

### Methods

1. Action: constructor, adds a handler to logger
2. dispose: dispose the action delegate
3. init: initializes this action delegate with the workbench window it will work in.
4. selectionChanged: Notifies the action delegate that the selection in the workbench has changed.
5. run: performs this action
6. getWindow: return the current WorkbenchWindow
7. findConsole: find current console, if there is no console available, create a new one.

### Attributes

1. log: final and static variable, represents a log
2. window: current workBenchWindow
3. uInteraction: an instance of UserInteraction class,
4. generateGML: an instance of GenerateGML class
5. CONSOLE: name of the console
6. console: an console instance

---

**GenerateGML**

```java
public class GenerateGML
{
  //currentProject: IProject
  //visitor: ASTVisitor
  //listener: VizzBackEnd
  //folderName: String
  //vecHeader: Vector<ITranslationUnit>
  //vecSource: Vector<ITranslationUnit>
  //CONSOLE: String = VizzAnalyzer
  //static final
  //console: MessageConsoleStream = findConsole(CONSOLE).newMessageStream
  //final

  +generate()
  +dfe(xco:ICElement)
  -visit()
  -findConsole(name: String): MessageConsole
}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>GenerateGML</td>
</tr>
<tr>
<td>Package in</td>
<td>package 2.1 Actions</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Get current C or C++ programs information in current work bench window.</td>
</tr>
<tr>
<td></td>
<td>Visit source files, after visiting, generate a GML file.</td>
</tr>
<tr>
<td>Relations</td>
<td>Used by Action</td>
</tr>
<tr>
<td></td>
<td>Use EclipseCFrontEnd &amp; EclipseCppFrontEnd</td>
</tr>
<tr>
<td>Methods</td>
<td>1. generate: get a C or C++ project, initial a</td>
</tr>
</tbody>
</table>
DirectedGraphInterface. Invoke method to traverse all source files. After traversing, create a GML file.

2. def: recursive method, separate headers and source files. Put header files into vecHeader Vector, source files vecSource Vector individually.

3. visit: recursive method, firstly visit all header files, then all source files.

4. findConsole: find current console, if no console is available, create a new one.

### Attributes

1. currentProject: the current C or C++ project
2. visitor: an instance of EclipseCFrontEnd or EclipseCppFrontEnd, which will be used to traverse a TranslationUnit
3. folderName: the folder name of current project
4. vecHeader: a Vector<String> which contains all header files of current project.
5. vecSource: a Vector<String> which contains all source files of current project.
6. CONSOLE: console name is “VizzAnalyzer”
7. console: an console instance

### UserInteraction

#### UserInteraction

```java
//CONSOLE: String = VizzAnalyzer
decl final
-console: MessageConsoleStream = findConsole(CONSOLE).newMessageStream

+print2console()
+frontEndDialog(window:IWorkbenchWindow)
-findConsole(name:String): MessageConsole
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>UserInteraction</td>
</tr>
<tr>
<td>Package in</td>
<td>package 2.1 Actions</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Deal with interaction with user</td>
</tr>
<tr>
<td>Relations</td>
<td>Used by Action</td>
</tr>
</tbody>
</table>
| Methods    | 1. print2console: print information to console  
             2. frontEndDialog: pop up a dialog to user  
             3. findConsole: find current console, if no console is available, create a new one. |
| Attributes | 1. CONSOLE: console name is “VizzAnalyzer”  
             2. console: an console instance |
6. Design
The chapter covers three design issues. The first section explains how the mapping between C/C++ specific Meta-Model and Common-Meta-Model is constructed. In the second section, key generating algorithm is described. Further, in the last section, we give the basic steps to be followed to create nodes and edges.

6.1 Meta-Model Mapping
This section discusses how C/C++ Specific Meta-model maps to Common-meta-model; especially, how C/C++ Meta-model in Eclipse CDT maps to CMM.

6.1.1 Common-Meta-Model
In the beginning, we introduce Common-Meta-Model in the representation of tree grammar and relations. Tree grammars for entities with their structural containment and relations describe relationship between those entities. Therefore, the common-meta-model can be denoted as $M = (G, R)$.

Table 6-1 shows Production and specialization hierarchy in CMM. Model is the root node type in CMM tree structure. The productions $P$ of $G$ describe a structural containment relation, denoted by ::= here, and a specialization hierarchy on entities, denoted by $\rightarrow$.

From the Table 6-1, we can find some entities denoted by “A<entity name>“. These entities are abstract model elements which could not used to instantiate a concrete model.

Furthermore, for some model element, there are unary relations. Only two kinds of unary relation is used in our current mapping which are $isConstructor$ ($Method$) and $InheritedFrom.inheritanceType$.

<table>
<thead>
<tr>
<th>Productions P and specialization hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model $::=\ AModelObject^*$</td>
</tr>
<tr>
<td>$\rightarrow\ AModelObject</td>
</tr>
<tr>
<td>$\rightarrow\ Package</td>
</tr>
<tr>
<td>$\rightarrow\ CompilationUnit$</td>
</tr>
<tr>
<td>$\rightarrow\ AType</td>
</tr>
<tr>
<td>$\rightarrow\ Type</td>
</tr>
<tr>
<td>$\rightarrow\ StructuredType</td>
</tr>
<tr>
<td>$\rightarrow\ AbstractClass</td>
</tr>
<tr>
<td>$\rightarrow\ Value</td>
</tr>
<tr>
<td>$\rightarrow\ Variable</td>
</tr>
<tr>
<td>$\rightarrow\ EnumerationLiteral^*$</td>
</tr>
<tr>
<td>$\rightarrow\ ExecutableValue</td>
</tr>
<tr>
<td>$\rightarrow\ FormalParameter^*$</td>
</tr>
<tr>
<td>Routine $::=\ FormalParameter^*$</td>
</tr>
<tr>
<td>Method $::=\ FormalParameter<em>Statement^</em>$</td>
</tr>
<tr>
<td>Statement $\rightarrow\ Loop</td>
</tr>
<tr>
<td>Loop $\rightarrow\ Do</td>
</tr>
<tr>
<td>Condition $\rightarrow\ If</td>
</tr>
<tr>
<td>Do $::=\ Statement^*$</td>
</tr>
<tr>
<td>For $::=\ Statement^*$</td>
</tr>
<tr>
<td>If $::=\ Statement^*$</td>
</tr>
<tr>
<td>Switch $::=\ Statement^*$</td>
</tr>
<tr>
<td>While $::=\ Statement^*$</td>
</tr>
</tbody>
</table>
Table 6-1: CMM, productions and specialization hierarchy

<table>
<thead>
<tr>
<th>Binary semantic relations in R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains                      : SourceObject × SourcePart, Package × ModelElement</td>
</tr>
<tr>
<td>HasValue                      : Variable × Value</td>
</tr>
<tr>
<td>Imports                       : Class × Package</td>
</tr>
<tr>
<td>InheritsFrom                 : Class × Class</td>
</tr>
<tr>
<td>Invokes                       : ABehaviouralElement × ABehaviouralElement</td>
</tr>
<tr>
<td>IsActualParameterOf           : ModelElement × Invokes</td>
</tr>
<tr>
<td>IsConstructorOf              : Class × Constructor</td>
</tr>
<tr>
<td>IsDefinedInTermsOf            : Type × Type</td>
</tr>
<tr>
<td>IsFieldOf                     : Field × StructuredType</td>
</tr>
<tr>
<td>IsMethodOf                    : Method × Class</td>
</tr>
<tr>
<td>IsOfType                      : Value × Type</td>
</tr>
<tr>
<td>IsSubPackageOf                : Package × Package</td>
</tr>
</tbody>
</table>

Table 6-2: Binary semantic relations in Common-meta-model

<table>
<thead>
<tr>
<th>Specialization hierarchy ⊃ of semantic relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains</td>
</tr>
<tr>
<td>Relationship</td>
</tr>
<tr>
<td>ModelRelationship</td>
</tr>
<tr>
<td>InheritsFrom</td>
</tr>
<tr>
<td>Invokes</td>
</tr>
<tr>
<td>IsPartOf</td>
</tr>
<tr>
<td>IsPartOfSignatureOf</td>
</tr>
</tbody>
</table>

Table 6-3: Specialization hierarchy of semantic relations

Table 6-2 shows how the relations in CMM are defined. For example, Invokes: ABehaviouralElement × ABehaviouralElement means that relation “Invokes” happens between two ABehaviouralElement model elements. Table 6-3 is specialization hierarchy of semantic relations in CMM.

Common-Meta-Model will be used as basis of our C/C++ Specific Meta-Model.

6.1.2 C/C++ Specific Meta-Model

This section describes the way we define C/C++ Front-end meta-model and the mapping functions between C/C++ specific front-end to Common-meta-model. We define the C/C++ Meta-Model first and then present the mapping functions between C/C++ specific Meta-Model to Common-Meta-Model first.

<table>
<thead>
<tr>
<th>Productions P and specialization hierarchy in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program^c : TranslationUnit'^c*</td>
</tr>
<tr>
<td>TranslationUnit^c : Function'^c*</td>
</tr>
<tr>
<td>Function^c : Parameter^c<em>Statement'^c</em></td>
</tr>
<tr>
<td>AVariable^c : Variable^c<em>Parameter'^c</em></td>
</tr>
<tr>
<td>AStatement^c : Do'^c* For'^c* Switch'^c<em>While'^c</em> If'^c*</td>
</tr>
<tr>
<td>Do^c : Statement'^c*</td>
</tr>
<tr>
<td>For^c : Statement'^c*</td>
</tr>
<tr>
<td>If^c : Statement'^c*</td>
</tr>
<tr>
<td>Switch^c : Statement'^c*</td>
</tr>
</tbody>
</table>
while<sup>C</sup> ::= Statement<sup>C</sup>*

Table 6-4: Productions and Specialization hierarchy in C

<table>
<thead>
<tr>
<th>Binary semantic relations in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes&lt;sup&gt;C&lt;/sup&gt;</td>
</tr>
<tr>
<td>Accesses&lt;sup&gt;C&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Table 6-5: Binary semantic Relations in C

<table>
<thead>
<tr>
<th>Mapping Functions α&lt;sup&gt;C&lt;/sup&gt; between C Meta-Model and Common-Meta-Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(program&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(translation_unit&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(Function&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(Variable&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(Do&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(For&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(If&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(Switch&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(While&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(Invokes&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;C&lt;/sup&gt;(Creates&lt;sup&gt;C&lt;/sup&gt;)</td>
</tr>
</tbody>
</table>

Table 6-6: Mapping Function α<sub>cpp</sub> between C Meta-Model and CMM

Table 6-4 defines production and the specialization hierarchy of these productions in C; notations used here have the same meaning as in Table 6-1.

Table 6-5 specifies two types of relations between productions. Because there is no specialization hierarchy in current C specific model, we don’t have corresponding table for C to table 6-3.

Table 6-6 illustrates the mapping between C specific Model and Common-meta-model.

Current C Specific Model doesn’t reflect all the C language specification. We build it according to requirements from certain metrics. This model can be easily extended.

Table 6-7, Table 6-8, Table 6-9 together define C++ Meta-Model and mapping functions to Common-Meta-Model. C++ Meta-Model can be regarded as superset of C Meta-Model which added some elements related to object-oriented language specification, such as class, constructor and field.

Table 6-4 defines production and the specialization hierarchy of these productions in C; notations used here have the same meaning as in Table 6-1.

Table 6-5 specifies two types of relations between productions. Because there is no specialization hierarchy in current C specific model, we don’t have corresponding table for C to table 6-3.

Table 6-6 illustrates the mapping between C specific Model and Common-meta-model.

Current C Specific Model doesn’t reflect all the C language specification. We build it according to requirements from certain metrics. This model can be easily extended.

Table 6-7, Table 6-8, Table 6-9 together define C++ Meta-Model and mapping functions to Common-Meta-Model. C++ Meta-Model can be regarded as superset of C Meta-Model which added some elements related to object-oriented language specification, such as class, constructor and field.
Table 6-7: Productions and Specialization hierarchy in C++

<table>
<thead>
<tr>
<th>Binary semantic relations in C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>InheritsFrom&lt;sup&gt;pp&lt;/sup&gt;       : Class&lt;sup&gt;pp&lt;/sup&gt; × Class&lt;sup&gt;pp&lt;/sup&gt;</td>
</tr>
<tr>
<td>Invokes&lt;sup&gt;pp&lt;/sup&gt;            : ABehaviouralElement&lt;sup&gt;pp&lt;/sup&gt; × ABehaviouralElement&lt;sup&gt;pp&lt;/sup&gt;</td>
</tr>
<tr>
<td>Creates&lt;sup&gt;pp&lt;/sup&gt;            : ABehaviouralElement&lt;sup&gt;pp&lt;/sup&gt; × Constructor&lt;sup&gt;pp&lt;/sup&gt;</td>
</tr>
<tr>
<td>Accesses&lt;sup&gt;pp&lt;/sup&gt;           : ABehaviouralElement&lt;sup&gt;pp&lt;/sup&gt; × Field&lt;sup&gt;pp&lt;/sup&gt;</td>
</tr>
<tr>
<td>: ABehaviouralElement&lt;sup&gt;pp&lt;/sup&gt; × Variable&lt;sup&gt;pp&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Table 6-8: Binary semantic Relations in C++

<table>
<thead>
<tr>
<th>Mapping Functions α&lt;sup&gt;pp&lt;/sup&gt; between C++ Meta-Model and Common-Meta-Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(program&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(translation_unit&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(class&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(constructor&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Function&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Variable&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Field&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Do&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(For&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(If&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Switch&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(While&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(InheritsFrom&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Invokes&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Creates&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
<tr>
<td>α&lt;sup&gt;pp&lt;/sup&gt;(Accesses&lt;sup&gt;pp&lt;/sup&gt;)</td>
</tr>
</tbody>
</table>

Table 6-9: Mapping Function α<sup>pp</sup> between C++ Meta-Model and CMM

6.1.3 Mapping CDT ASTNode to C/C++ Model Element

Because we use AST created by Eclipse CDT as input to our C/C++ Front-end, we need to map CDT ASTNode types to C/C++ Model element. Table 6-10 shows the mapping functions for C and Table 6-11 for C++. Besides, for some of the elements, we have further explanations to say why we use certain kind of ASTNode in the mapping function and how to use those ASTNodes. We select five kinds of elements here, class, function, constructor, variable and field. Since C Meta-Model is a subset of
C++ Meta-Model, we choose elements in C++ Meta-Model as examples to show our explanation.

Class\textsuperscript{cpp}:

\textit{Description}: Represents a class declaration in a program.

\textit{ASTNode type in CDT}:

a) Type name: CPPASTCompositeTypeSpecifier

b) Code:

\begin{verbatim}
if (declSpec instanceof CPPASTCompositeTypeSpecifier) {
    CPPASTCompositeTypeSpecifier element =
    (CPPASTCompositeTypeSpecifier) declSpec;
    if (element.getKey() == element.k_class) {
        ...
    }
}
\end{verbatim}

i. About getKey(): getKey() method returns an int value defined in interface IASTCompositeTypeSpecifier. Since there are several kinds of “Composite Type” in C (struct & union) and C++ (struct, union & class). This Key value will be used to distinguish whether the node is a struct, union or class.

c) Path in CDT:

org.eclipse.cdt.internal.core.dom.parser.c.CPPASTCompositeTypeSpecifier

### Mapping Functions $\alpha^{cpp}$

\begin{align*}
\alpha^{cpp}(ICModel) & \mapsto \text{program}^{cpp} \\
\alpha^{cpp}(CPPASTTranslationUnit) & \mapsto \text{translation_unit}^{cpp} \\
\alpha^{cpp}(CPPASTCompositeTypeSpecifier (element.getKey == element.k_class)) & \mapsto \text{class}^{cpp} \\
\alpha^{cpp}(CPPASTFunctionDefinition) & \mapsto \text{constructor}^{cpp} \\
\alpha^{cpp}(CPPASTFunctionDefinition (see explanation in Constructor part)) & \mapsto \text{Function}^{cpp} \\
\alpha^{cpp}(CPPASTName (binding == CPPVariable && binding != CPPField)) & \mapsto \text{Variable}^{cpp} \\
\alpha^{cpp}(CPPASTName (binding == CPPField)) & \mapsto \text{Field}^{cpp} \\
\alpha^{cpp}(CPPASTDoStatement) & \mapsto \text{Do}^{cpp} \\
\alpha^{cpp}(CPPASTForStatement) & \mapsto \text{For}^{cpp} \\
\alpha^{cpp}(CPPASTIfStatement) & \mapsto \text{If}^{cpp} \\
\alpha^{cpp}(CPPASTSwitchStatement) & \mapsto \text{Switch}^{cpp} \\
\alpha^{cpp}(CPPASTWhileStatement) & \mapsto \text{While}^{cpp}
\end{align*}

Table 6-11: Mapping between C++ Meta-Model and CDT ASTNodes

Function\textsuperscript{cpp}:

\textit{ASTNode type in CDT}:

a) Type name: CPPASTFunctionDefinition
b) The reason for using CPPASTFunctionDefinition:
   i. There is another ASTNode in CDT named “CPPASTFunctionDeclarator”. Here we choose CPPASTFunctionDefinition, because CPPASTFunctionDeclarator is only declaration of function, from which we cannot get context in one function which is needed during analysis.
   ii. From CPPASTFunctionDefinition, we can always get a corresponding CPPASTFunctionDeclarator node. Then we can get parameter information of a function from its declarator.
   iii. In C and C++, function can be declared for several times, so in one program, there could be many CPPASTFunctionDeclarator but one CPPASTFunctionDefinition for one function.

c) Path in CDT:
   org.eclipse.cdt.internal.core.dom.parser.c.CPPASTFunctionDefinition

Constructor<cpp>:

ASTNode type in CDT:

a) Type name: CPPASTFunctionDefinition
b) Separate “Constructor” from other function definitions

```cpp
#include "Hello.h"

class Hello2: public Hello {

public:
    int hello1;

public:
    Hello2()
    {
        hello1 = 6;
    }

public:
    ~Hello2();

public:
    Hello2(int i) {
        hello1 = i;
    }

public:
    int getHello(Hello * h);
};
```

Figure 6-1: Example of constructor

i. We can get “binding types” of a ASTNode using the ASTName of it.
ii. There are several binding types for CPPASTFunctionDefinition
   i. CPPConstructor
   ii. CPPImplicitConstructor
   iii. CPPFunction
iii. Definition of class member function can be written either in the same file of the class or in another .cpp file. Figure 6-1 gives an example to show when the binding types are.
   i. Only in the case “Hello2()” in Figure 6-1(inside a class, without
parameters), binding is CPPImplicitConstructor

ii. Only in the case “Hello2(int i)” (inside a class, with some parameters), binding is CPPConstructor

iii. Otherwise, binding of constructor will be “CPPFunction”. To find all the constructors, some filter should be applied in the case “binding instanceof CPPFunction”

c) Path in CDT:
org.eclipse.cdt.internal.core.dom.parser.c.CPPASTFunctionDefinition

Variable<sup>CPP</sup>:
**ASTNode type in CDT:**

a) Type name: CPPASTName

b) The way to get Variable in CPPASTName:
   Get binding of CPPASTName. Because CPPField is a sub-class of CPPVariable, we should separate CPPField from CPPVariable. For doing this, we need to check whether the binding is a CPPField first. If binding is not a CPPField, the CPPASTName is a Variable.

c) Path in CDT:
org.eclipse.cdt.internal.core.dom.parser.c.CPPASTName

Field<sup>CPP</sup>:
**ASTNode type in CDT:**

a) Type name: CPPASTName

b) The way to get Field in CPPASTName:
   Get binding of CPPASTName. If binding is a CPPField, the CPPASTName is a Field.

c) Path in CDT:
org.eclipse.cdt.internal.core.dom.parser.c.CPPASTName

### 6.2 Node Key Generating

We are aware that for creating edges, some nodes need to have unique keys. However, there is no suitable keys provided by CDT, we need to generate the keys by ourselves. Since such keys are available in JDT, we will take the key generating algorithm in JDT as reference.

#### 6.2.1 Key Generating Algorithm in JDT

The key generating algorithm is from JDT API for method org.eclipse.jdt.core.dom.IBinding getKey().

1. packages - the name of the package (for an unnamed package, some internal id)
2. classes or interfaces - the VM name of the type and the key of its package
3. array types - the key of the component type and number of dimensions
4. primitive types - the name of the primitive type
5. fields - the name of the field and the key of its declaring type
6. methods - the name of the method, the key of its declaring type, and the keys of the parameter types
7. constructors - the key of its declaring class, and the keys of the parameter types

Table 6-12: Key Generating Algorithm in JDT<sup>[15]</sup>
### 6.2.2 Key Generating Algorithm in C/C++

Table 6-8 shows how keys are generated in C/C++. Five kinds of elements are needed to have unique keys, which are class, function, constructor, variable and field. We organize the key generating algorithm for C/C++ together, but actually, for C, only function and variable make sense. The other three kinds of elements are C++ specific.

- **Class** – For C/C++, there is no corresponding concept to “package”, so we use file name instead of “package key”. Because the file name is full path of a file, it can be used to distinguish different files with the same name but located in different places.
- **Function** – In Java, all methods are inside classes, but in C/C++, it is not in the case. Thus, we use full file name instead of “The key of its declaring type”.
- **Constructor** – In C/C++, constructor can be defined outside declaration of the class it belongs to. Therefore, we regard constructor as a special kind of function, using similar way to generate keys.

<table>
<thead>
<tr>
<th>Key Generating Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
</tr>
<tr>
<td><strong>Function</strong></td>
</tr>
<tr>
<td><strong>Constructor</strong></td>
</tr>
<tr>
<td><strong>Variable</strong></td>
</tr>
<tr>
<td><strong>Field</strong></td>
</tr>
</tbody>
</table>

Table 6-13: Key Generating Algorithm in C/C++

- **Variable** – We only take local variable into consideration, so we can take advantage of the function key the variable belongs to.
- **Field**: using the same way as in Java

Table 6-12 shows the Key generating algorithm in C/C++ according to our discussion above.

### 6.2.3 Example of Key Generating Algorithm in C++

This section provides an example which is used to show the keys generated in practice.

```cpp
#include "ExampleClass.h"
#include <iostream>
using namespace std;

int main() {
    ExampleClass * ec = new ExampleClass();
    int b = ec-> getExample("Hello");
    return b;
}
```

Figure 6-2: Generating key Example Program -- main.cpp

```cpp
#include "ExampleClass.h"
#include <string>
using namespace std;

ExampleClass::ExampleClass() {
    example = 5;
}

int ExampleClass::getExample(string s) {
    return example;
}
```

Figure 6-3: Generating key Example Program -- ExampleClass.cpp

```cpp
#include <string>
```


```
using namespace std;

#pragma once

class ExampleClass {
private:
  int example;
public:
  ExampleClass();
  public:
  int getExample (string s);
};
```

**Figure 6-4: Generating key Example Program -- ExampleClass.h**

<table>
<thead>
<tr>
<th>Key Generating Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>Constructor</td>
</tr>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>Field</td>
</tr>
</tbody>
</table>

Table 6-14: Example key generating result

This example has three source files including a file with main function (as Figure 6.3) and a class .cpp file (as Figure 6.4) and a class .h file (as Figure 6.5). We assume all the files are directly under C disk. Example result can be found in Table 6-13.

### 6.3 Steps of Creating Node and Edge

This section gives an overview about the basic steps we should follow to create node and edge. The procedure can be described briefly as following:

**Creating a node:**
1. Find the ASTNode which presents one kind of meta-model element or separate meta-model element from one kind of ASTNode. (as discussed in section 6.1.3)
2. Generate key if necessary (for class, function, constructor, variable and field as discussed in section 6.2.2)
3. Add a node (with available information, such as key)

**Creating an edge:**
1. Decide when to create an edge
2. Generate the key for target element
3. Add an edge.
7. Implementation
This Chapter covers some implementation issues of C/C++ Front-End. Because it is not possible to mention all the implementation detail here, we organized implementation into four aspects. For each aspect, we select the most necessary and interesting detail to show here. First of all is start-up part explaining the way to create a working environment of C/C++ Front-End. Then, we explain how to get C/C++ source files to traverse. In the end, implementation detail for creating each node and edge is discussed in the last two sections.

7.1 Start-up procedure
This section explains how to create Eclipse plug-in environment for further development of C/C++ front-end.

7.1.1 Get and install Eclipse CDT:
Here are the steps to get Eclipse CDT:
1. Download the latest milestone of CDT: CDT 4.0.0Build RC2 on http://download.eclipse.org/tools/cdt/releases/europa/dist/4.0.0RC2/, this version of CDT requires Eclipse 3.3M6+.
2. Copy file in two folders: plugins and features in CDT to corresponding folders under Eclipse home folder.
3. Restart the Eclipse.

7.1.2 Create a new Plug-in project:
C/C++ Front-end works as a Eclipse Plug-in project. We create a new menu item for invoking methods in C/C++ Front-end. Here are the steps for creating such a project. We take Java front-end as reference when we create the plug-in project.

Steps as following:
1. Create a new Plug-in project in Eclipse named “C_CppFrontEnd”
2. Add three dependencies in “Dependencies” page
   a. org.eclipse.core.resources
   b. org.eclipse.cdt.core
   c. org.eclipse.ui.console
3. Add four external jar files
   a. create a folder “lib” under folder “C_CppFrontEnd”
   b. put grail, backend, attribute, model (explained in section 5.3) into “lib”
   c. in “runtime” page, add four jar files in “lib” into classpath
4. Add one extension in “extensions” page
   a. Extension: org.eclipse.ui.actionSets
   b. Description: This extension point is used to add menus, menu items and toolbar buttons to the common areas in the Workbenchwindow.
5. Rewrite plugin.xml:
   a. Usage of this xml file is to create a menu item with the name “Retrieve Data Action”, this action point is related to class “vizzanalyzer.frontends.CandCpp.eclipsePlugin.actions.Action”.

35
7.2 Get Source Files

The “Get Source Files” part mainly in GenerateGML class.

We design to traverse each opened project twice for separating header files and source files. In the first time we get the type (header file or source file) of each file in the project. If it is a header file, put it into a Vector for header files; otherwise, put it into a Vector for source files. The second time, we traverse files in Vector <header> first, afterwards traversing files in Vector <source>.

The reason for doing this is to identify a “field” node. Because in C++ program language, class declaration and class realization can be separated in two files. If they are separated, we will not be able to identify whether a variable in a class-member function is actually a field or not. Thus, we need to go back to check the class declaration. In the case we don’t separate header file, source file and to traverse header file first. The class declaration still doesn’t exist at that time.

Besides separating header files and source files, we also need to decide whether a file is a C file or C++ file, because these two kinds of files need different traversing method. (EclipseCFrontEnd or EclipseCppFrontEnd).

The traversing of a project is done using a recursive method “dfs”. When it comes a ITranslationUnit node, it will invoke method to traverse files.

7.3 Generate Key

In section 6.2.2, we mentioned that 5 kinds of ASTNodes need to have unique keys and presented key generating algorithm for each of them. In this section, we use constructor key as example to show the process of generating a key.

Constructor key consists three parts of information:
1. Full file name
2. Constructor name
3. Parameter types name list

The first two ones are easy to get as “fileName” and “functionName” following:

```java
else if (declaration instanceof CPPASTFunctionDefinition) {
    CPPASTFunctionDefinition element =
        (CPPASTFunctionDefinition) declaration;
    IASTName name = element.getDeclarator().getName();
    String fileName = null;
    fileName = element.getFileLocation().getFileName();
    functionName = name.toString();
    ...... 
}
```

The most challenge work is to get parameter type list.

```java
CPPASTFunctionDeclarator declarator =
    (CPPASTFunctionDeclarator)element.getDeclarator();
IASTParameterDeclaration[] parameters = declarator.getParameters();
for (int i = 0; i < parameters.length; i++) {
    String temp = null;
    if (parameters[i] instanceof CPPASTParameterDeclaration) {
        IASTDeclSpecifier type = parameters[i].getDeclSpecifier();
        if (type instanceof CPPASTSimpleDeclSpecifier) {
            CPPASTSimpleDeclSpecifier simpleType =
                (CPPASTSimpleDeclSpecifier) type;
            int iSimpleType = simpleType.getType();
        }
    }
}
```
switch (iSimpleType) {
    case 2:
        temp = "char";
        break;
    case 3:
        temp = "int";
        break;
    case 4:
        temp = "float";
        break;
    case 6:
        temp = "bool";
        break;
    default:
        temp = "other";
        break;
}
}

else if (type instanceof CPPASTNamedTypeSpecifier) {
    CPPASTNamedTypeSpecifier namedType =
        (CPPASTNamedTypeSpecifier) type;
    temp = namedType.getName().toString();
} else {
    temp = "unknown"
}

parameterTypes = parameterTypes + temp;
parameterTypes = "(" + parameterTypes + ");"

We can get a parameter list from function declaration (line 3), go through the list, get type Specifier of each parameter (line 7). If the type is a CPPASTSimpleDeclSpecifier, we go into it to see whether it is a “char”, “int”, “float” or “bool”. If the type is a CPPASTNamedTypeSpecifier (predefined types in C/C++, such as “string”, or used defined new class type), we get the name of the type.

We treat CPPASTSimpleDeclSpecifier in this way, because the limitation we meet during creating edge. See section 7.5 for detailed information.

7.4 Create a Node
We described in section 6.3 the basic steps of creating a node. Here we use “constructor” as example to show how these steps are processed.

1. Find the ASTNode which presents one kind of meta-model element or separate meta-model element from one kind of ASTNode.
   As we discussed in Section 6.1.3, constructor implementation may be found in two places. One is inside declaration of class; the other is outside the declaration.

Case 1:
if (binding instanceof CPPConstructor
    || binding instanceof CPPImplicitConstructor) {
    ……
}

Case 2:
else if (binding instanceof CPPFunction) {
    if (name instanceof CPPASTQualifiedName) {
        CPPASTQualifiedName qualifiedName =
            (CPPASTQualifiedName) name;
        IASTName[] names = qualifiedName.getNames();
        if (names.length == 2 &&
In the second case, we compare whether the name of this function qualified name with two simple names and these two simple names are identical. If the answer is yes, it is a constructor.

2. Generate key if necessary
   As we explained in last section

3. Add a node (with available information, such as key)

   ```java
   listener.enterNode(CommonMetaModel10.Constructor,
   constructorKey, "Cons:" + element.getDeclarator().getName());
   ```

In this part, we use event-based architecture. We notified the listener of back-end that we have entered a new node.

### 7.5 Create a Edge

In section 6.3, we also describe the basic steps of creating an edge. In this section, we use “invokes” edge to show how we follow the steps to create an edge.

1. Decide when to create an edge
   There is a `CPPASTFunctionCallExpression` node, which can be used as the point to create “invokes” edge.

2. Generate the key for target element
   Firstly, we need to get parameter types list. We create a private method which is used to generate parameter types list -- `getParameterTypes`.
   
   A function definition will have a list of IASTExpression which contains parameter information. But we cannot get parameter type directly from the list. Instead, we will get the parameter value. One way to get parameter types is to get the function declaration, however, we can only use name of function to get

   ```java
   private void getParameterTypes(IASTExpression expression) {
     if (expression == null) {
     } else if (expression instanceof IASTExpressionList) {
       IASTExpressionList eList = (IASTExpressionList) expression;
       IASTExpression[] e = eList.getExpressions();
       for (int i = 0; i < e.length; i++) {
         getParameterTypes(e[i]);
       }
     } else if (expression instanceof IASTLiteralExpression) {
       IASTLiteralExpression eLiteral = (IASTLiteralExpression) expression;
       Integer i = new Integer(eLiteral.getKind());
       switch (i) {
         case 0:
           parameterTypes = parameterTypes + "int";
           break;
         case 1:
           ```
function declaration. When there is overloading existing, the answer will not be right. Therefore, we have to use parameter value to get its type. “getParameterTypes” is a recursive method, if the expression turns out to be a “express list”, the method will go on exam its child. One possibility is that the value is a literal value. We get type from IASTLiteralExpression. In section 7.3, we get the kind of simple type from IASTSimpleDeclSpecifier, which uses different integer to present type from IASTLiteralExpression. Thus, we need to convert those integer such as “0”, “1” into “int”, “float” to make the type value consistent. In the case of “expression instanceof IASTIdExpression”, we use similar way to section 7.3 to get the parameter types.

Two kinds of “invokes” in C++, one is invokes a class-member function using field reference, while the other is to invoke a function defined outside a class. These two kinds of “invokes” should be treated in different ways.

Case 1: Call a class-member function

The basic idea to get the key in this case is also to get fileName, functionName, and parameterTypeList of target function.
functionName: To get the functionName is the major problem in this case. Since we can only get a the function Name itself, but actually for a class-member function, class name is crucial. Here the functionName means the combination of className and function name itself. We need to go back to see the declaration type of the class variable (using the classVariable, it calls the class-member function). And then get class name. At last, put className and function name together to get the functionName.

fileName: one problem is the target function is not in the same file of current call site. We store class-member function names in Statistics class. When we need to get the IASTName of a function using the functionName mentioned above, we look up in Statistics. From the IASTName node of the target function, we can get its file information.

parameterTypeList: when we try to find the declaration of class variable, we will come across a NewExpression node, from this node, it will be possible to get parameter expression information. Using “getParameterTypes” method, we will get the parameterTypeList.

Put the three values together, we can get the target function key successfully.

Case 2: Call a function which is not a class-member function.

```
1   else if (expression instanceof CPPASTFunctionCallExpression) {
2     CPPASTFunctionCallExpression element =
3         (CPPASTFunctionCallExpression) expression;

4     IASTExpression expre = element.getFunctionNameExpression();

5     else if (expre instanceof CPPASTIdExpression) {
6         IASTExpression parameterExpr =
7             element.getParameterExpression();
8         parameterTypes = new String();
9         getParameterTypes(parameterExpr);
10        parameterTypes = "(" + parameterTypes + ")";
11        CPPASTIdExpression id = (CPPASTIdExpression) expre;
12        functionName = id.getName().toString();
13        functionKey = fileName + functionName + parameterTypes;
14    }
15 }
```

We can get the function name from the CPPASTIdExpression (line 12). Using “getParameterTypes”, we can get the parameter types list. Together with filename, it is enough to get the target key which will be used to create the edge.

3. Add an edge.

```
1   Attributes a = new Attributes(
2     CommonMetaModel10.name,
3     (debug ? "Function:" : functionName));
4   listener.addCrossReferenceEdge(
5     CommonMetaModel10.Invokes, element,
6     functionKey, a.getAttributes());
7     parameterTypes = "";
```

This part uses event-base architecture. Notify back-end listener, we want to add another “invokes” edge. When we add the edge, we can also specify some
attributes of the edge.

To create an edge, we need to get source key and target key. In the case, source key is the element itself, and target key has been generated in last step.
8. Conclusion and Future Work

This last chapter draws conclusion of this thesis, and points out the directions of future work. First, in this conclusion section, we look back the problems and goals in the introduction to see whether we have achieved our goals according to the criteria. In the future work section, we will give some ideas about how to improve current implementation of C/C++ Front-end.

8.1 Conclusions

In the introduction chapter, we summarized the problem of this thesis as:

**The problem tackled by this thesis is to define C/C++ front-end allowing analyzing C/C++ program in the VizzAnalyzer.**

The motivation for doing this is that before having the C/C++ front-end, VizzAnalyzer can only analyze Java and UML programs. Considering that C and C++ are two widely used programming languages, it is a valuable work to bring VizzAnalyzer to C/C++ domain.

To solve the problem, we gave four major goals of this thesis and corresponding criteria for each goal used to check whether we have achieved the goal or not.

The first goal is to define C/C++ front-end Meta-Model. This goal has been realized since we use Eclipse CDT to parse opened C/C++ programs in WorkBenchWindow, and get C/C++ front-end Meta-Model from it. Detailed implementation is stated in Section 7.2.

The second goal is to define mapping between C/C++ specific front-end Meta-Model and Common-Meta-Model. We present the Common-Meta-Model in section 6.1. Then in section 6.2, we give C/C++ Meta-Model along with mapping functions to Common-Meta-Model. Especially, since we use Eclipse CDT as the compiler to get C/C++ specific front-end, we need to map Eclipse CDT ASTNode types to C/C++ Meta-Model elements. This step is achieved in section 6.3. Having these three sections (6.1, 6.2, 6.3), we can say the second goal has been reached. Additionally, implementation detail of how the mapping is realized can be found in Section 7.3.

The third goal is to use suitable infrastructure. In chapter 3 State of the Art, we selected Eclipse CDT as compiler for C/C++ programs. Eclipse CDT makes it possible for us to use Eclipse as infrastructure of C/C++ Front-End. Since we regard Eclipse is suitable for our project, this goal is achieved.

The last goal of our C/C++ Front-end is to use event-based architecture. As we described in section 5.1 C/C++ Meta-Model architecture, we can say this event-based architecture has been successfully integrated into our project. Therefore, the last goal has been realized.

The previous paragraphs show that all goals stated in the introduction chapter of this thesis have been successfully realized. We can make a conclusion that the problem of this thesis has been solved successfully.

Working on this thesis gives me a chance to have an insight of VizzAnalyzer and especially, Lincke’s work related to Software Quality Model. Before this thesis, I took a course Advanced Techniques for Program Analysis, I was very happy I could have the opportunity to put my theory knowledge into practice. After working on the thesis, I have a much deeper understanding about program analysis and the structure of VizzAnalyzer. Besides understanding the theoretical aspect of this thesis, another difficulty I met during the working process was to understand Eclipse CDT and to know how to take advantage of it. It was a good practice to work with existing code.
and learn how to get access to it in a quick way. Additionally, by working on the thesis, I got some knowledge about C/C++ language specification on the compiler level which was totally new for me before. Considering about this, I think I improved my learning ability in a large scale during the process working on the thesis.

8.2 Future work
Our current implementation of C/C++ Front-End provides us expected result and has already solved our problem. But there is still some room for improvement which can be done in the future work. Firstly, current version of Eclipse CDT has some limitation, for example, we need to generate unique keys for certain ASTNodes since there is no such key available in CDT now. Although we tried to ensure the key generated in a suitable way, it would be better to use the key provided by CDT to make the implementation more efficiently. Another thing needs to be taken into concern is that the C/C++ Meta-Model needs to be evolved according to our research in metrics analysis. If a new metrics needs more information from C/C++ Meta-Model, C/C++ Meta-Model should be extended to provide more information.
Reference

[1] Jussi Koskinen, Software Maintenance Costs,  
http://www.cs.jyu.fi/~koskinen/smcosts.htm  
Information Technology Research Institute, ELTIS-project,  
University of Jyväskylä.

[2] Rüdiger Lincke, Validation of a standard- and Metric-Based Software  
Quality Model, Växjö University

[3] David N. Welton, Programming Language Popularity,  
http://www.dedasys.com/articles/language_popularity.html

Analysis, Växjö University

for Program Analysis. In 22th IEEE International Conference on Software  

for Program Analysis. In 22th IEEE International Conference on Software  

Analysis, Växjö University

Växjö University, p15


[13] Rüdiger Lincke, Validation of a standard- and Metric-Based Software  
Quality Model, Växjö University pp 93–95

[14] Rüdiger Lincke, Validation of a standard- and Metric-Based Software  
Quality Model, Växjö University, pp 73–75

[15] Eclipse JDT API, IBinding, getKey().  