Using an XML-driven approach to create tools for program understanding.
An implementation for Configura and CET Designer

by

Åsa Wihlborg

LIU-IDA/LITH-EX-A--10/017--SE

2010-03-18
Final Thesis

Using an XML-driven approach to create tools for program understanding
An implementation for Configura and CET Designer

by

Åsa Wihlborg

LIU-IDA/LITH-EX-A--10/017--SE

2010-03-18

Supervisor: Peter Dalenius
IDA, Linköpings universitet
Emma Johansson
Configura Sverige AB

Examiner: Anders Haraldsson
IDA, Linköpings universitet
Titel: Ett XML-drivet tillvägagångssätt för att skapa vertyg för programförståelse

Using an XML-driven approach to create tools for program understanding.

Författare: Åsa Wihlborg

Sammanfattning:

A major problem during development and maintenance of software is lack of quality documentation. Many programmers have problems identifying which information is relevant for someone with no knowledge of the system and therefore write incomplete documentation. One way to get around these problems would be to use a tool that extracts information from both comments and the actual source code and presents the structure of the program visually.

This thesis aims to design an XML-driven system for the extraction and presentation of meta information about source code to that purpose. Relevant meta information in this case is, for example, which entities (classes, methods, variables, etc.) exist in the program and how they interact with each other.

The result is a prototype implemented to manage two company developed languages. The prototype demonstrates how the system can be implemented and show that the approach is scalable. The prototype is not suitable for commercial use due to its abstraction level, but with the help of qualified XML databases there are great possibilities to build a usable system using the same techniques in the future.
Abstract

A major problem during development and maintenance of software is lack of quality documentation. Many programmers have problems identifying which information is relevant for someone with no knowledge of the system and therefore write incomplete documentation. One way to get around these problems would be to use a tool that extracts information from both comments and the actual source code and presents the structure of the program visually.

This thesis aims to design an XML-driven system for the extraction and presentation of meta information about source code to that purpose. Relevant meta information in this case is, for example, which entities (classes, methods, variables, etc.) exist in the program and how they interact with each other.

The result is a prototype implemented to manage two company developed languages. The prototype demonstrates how the system can be implemented and show that the approach is scalable. The prototype is not suitable for commercial use due to its abstraction level, but with the help of qualified XML databases there are great possibilities to build a usable system using the same techniques in the future.

Sammanfattning

Ett stort problem under utvecklingen och underhållet av mjukvara är bristande dokumentation av källkoden. Många programmerare har svårt att identifiera vilken information som är viktig för någon som inte är insatt i systemet och skriver därför bristfällig dokumentation. Ett sätt att komma runt dessa problem skulle vara att använda verktyg som extraherar information från såväl kommentarer som faktisk källkod och presenterar programmets struktur på ett tydligt och visuellt sätt.

Det här examensarbetet ämnar att designa ett system för XML-driven extrahering och presentation av metainformation om källkoden med just det syftet. Metainformationen som avses här är exempelvis vilka entiteter (klasser, metoder, variabler, mm.) som finns i källkoden samt hur dessa interagerar med varandra.

Resultatet är en prototyp implementerad för att hantera tvåföretagsutvecklade språk. Prototypen demonstrerar hur systemet kan implementeras och visar att metoden är skalbar. Prototypen är abstraktionsmässigt inte lämplig för kommersiellt bruk men med hjälp av kvalificerade XML-databaser finns det stora möjligheter att i framtiden bygga ett praktiskt användbart system baserat på samma tekniker.
Acknowledgments

I would like to thank my supervisors Emma Johansson and Mikael Ågerud for their continuous support during the writing of this thesis. I would also like to thank Anders Haraldsson and Peter Dalenius at Linköpings universitet for their help and enthusiasm.
# Contents

1 **Introduction**  1
   1.1 Background .................................................. 2
   1.2 Purpose ...................................................... 2
   1.3 Goal .......................................................... 3
   1.4 Method ....................................................... 3
      1.4.1 Research ............................................... 3
      1.4.2 Implementation ....................................... 3
   1.5 Limitations .................................................. 3
   1.6 Thesis Outline .............................................. 4

2 **Background**  5
   2.1 Review of existing systems at Configura .................. 5
      2.1.1 Self developed programming language: CM ........... 5
      2.1.2 C with self developed pre-processor............... 6
      2.1.3 Development environment: Emacs .................... 7
      2.1.4 Handwritten Documentation ........................... 8
   2.2 Expectations and demands ................................. 8
      2.2.1 Management ............................................ 8
      2.2.2 Project managers and developers ................... 8
      2.2.3 R&D .................................................. 8

3 **Problem description**  11
   3.1 General problem description ............................... 11
   3.2 Relevant information to extract ........................... 12
      3.2.1 Explicit semantic information ....................... 12
      3.2.2 Implicit semantic information ...................... 13
      3.2.3 Explicit structural information .................... 13
      3.2.4 Implicit structural information .................... 13
      3.2.5 Macros ............................................... 13
   3.3 Extraction methods ......................................... 13
      3.3.1 Parser ............................................... 14
      3.3.2 Compilation output .................................. 14
      3.3.3 Querying an accessible Compiler ................... 14
   3.4 Previous research .......................................... 15
Chapter 2

Introduction

The most common meaning of the word documentation when referencing source code is either inline documentation, e.g. comments added to the source code documents or other types of handwritten documentation, for example UML diagrams, functional specifications or architectural views.

The need for documentation is currently under debate. There are many different views and little concrete research that provides any proof. Extreme programming enthusiasts claim there is little need for documentation during a software development as they rely on small teams and an informal knowledge base[5]. They do not however address the problem with how to maintain systems after the original creators have disappeared, or how to get new personnel to understand existing code and become familiar with the system. Others propose written documentation at different levels. Even though this theoretically solves the problem there are a number of practical problems such as getting developers to actually write it and then read it. Since documentation often is poorly written and not frequently updated programmers tend to distrust it which severely limits its usefulness.

Since documentation is traditionally written by hand and therefore relies on the programmer remembering to update the documentation when changing the source code, it is practically outdated as soon as it is written. This leads to a lot of misleading and false information. As a result the only thing that can be trusted to truly represent the structure of a program is its source code.

When learning or maintaining a system with little or no documentation the only reliable source of information is the source code. In projects with millions of lines of code, the task of understanding it by reading the code is virtually impossible. Even for smaller programs the process of reading source code is tedious and prone to error. When the programmer is familiar with the system and the overall structure, reading source code gives in depth and useful information.

Using inline comments is a very common way of documenting source code. The theory is that since the source code exists side-by-side with the documentation it will be easy to update the documentation when you change the code. This is not always the case however, and source code is often changed without the changes being reflected in the documentation. In that sense documentation
becomes obsolete as you write it. There is however no proof as to whether this kind of documentation is helpful.

One proposed solution to this problem is reverse engineering tools and other tools to help the programmer understand the code without relying on handwritten documentation. Most of these tools deal with software visualization [3] through clustering of software components [6, 17]

1.1 Background

Configura Sverige AB is the inventor of the concept of PGC (Parametrical Graphical Configuration). PGC is a technology and a business strategy for companies selling products that in some manner are configurable. The programs that Configura develops support configuring products in a 3D-environment and are integrated with the company’s business systems and provide support for everything from invoicing to tagging and assembly instructions. Figure 1.1 shows an image of a configured kitchen.

The solution Configura AB provides consists of a standard program and an extension developed specifically for every company. The extension describes the company’s product line and its business logic. All currently existing extensions have been developed by Configura, but there is no reason other companies could not develop extensions for Configura’s framework since Configura offers developer licenses. The problem is that there little documentation exists for either framework. The code has inline interface documentation but there is little more. Therefore all new developers have been educated internally at the company, mostly by sitting close to other developers and asking them when stuck. It is not possible to train a great number of partner developers this way.

Configura Sverige AB has two different frameworks: Configura and CET Designer. Configura is an old legacy system written in C with an early form of object orientation implemented with a preprocessor. CET Designer is implemented in CM which is a language developed by Configura AB especially for their applications.

Therefore Configura needs some kind of generated documentation or programming aiding tools to help facilitate the learning process among partner developers as well as new employees.

1.2 Purpose

The purpose of this thesis is to provide external developers working with the Configura or CET Designer platforms with tools to facilitate program understanding. The thesis examines how such a system could be designed and implemented.
1.3 Goal

The goal of this thesis is to present a design for a system for code understanding that is extendable and can handle several different languages and to develop a prototype that implements the core functionality and shows how the system could work.

1.4 Method

This thesis requires both research and implementation.

1.4.1 Research

The research necessary is twofold. Information about what the company needs and expects will be obtained through interviews with company staff. A review of current solutions and current research will be performed as well.

1.4.2 Implementation

A prototype will be implemented to prove that the solution is feasible. The implementation will by no means be a complete implementation but an important subset will be implemented to show certain features of the overall system.

1.5 Limitations

Since the time provided for this thesis is limited, the thesis will not try to answer the question of what kind of documentation and visualizations are useful for an objective but rather focus on how different types of documentation can be generated in a framework and how problems of scalability can be resolved. Whether or
not the generated documentation and views provided are truly useful can only be discovered through extensive observation and testing on several different types of users and is beyond the scope of this thesis.

Neither will optimization be a focus in this thesis. The approach needs to be fast enough for the solution be usable with the amount of data in question, but there will be no special consideration for the verbosity of the data format used or for the eventual diminished speed the application might suffer as a result.

1.6 Thesis Outline

Chapter 1: Introduction
Here I will elaborate on the goal for this thesis and explain the approach and limitations.

Chapter 2: Background
I will start by evaluating the system and development environment currently in use and then present the collected wishes and expectations expressed among company employees and managers regarding a complete solution.

Chapter 3: Problem description
Here I will explain the problem in detail and elaborate on certain interesting aspects. I will also summarize some previous research regarding similar topics, some general approaches and evaluate existing solutions who use them.

Chapter 4: Implementation
In the next chapter I will explain my general solution, describe my implementation in detail and then conclude with presenting the finished prototype.

Chapter 5: Conclusions
In the final chapter I will summarize my work and the conclusions that can be drawn. I will also suggest some future work.

Appendices
In the appendices I will show some code examples to illustrate my work more closely.
Chapter 2

Background

In this chapter I will provide my research into what is demanded and expected from a generated documentation system at Configura and what systems and environment are currently in use.

2.1 Review of existing systems at Configura

The development environment in use at Configura AB is Emacs. The languages used are CM and C. CET Designer is developed in CM and Configura in C with an early form of object orientation using a preprocessor and macros.

2.1.1 Self developed programming language: CM

CM is developed at Configura and specially designed for its applications, but with a syntax similar to Java and C#. CM was also developed to be a language which facilitates development by not only considering how a language should work for optimal efficiency when run, but also what makes a language enjoyable to use and a fast language to implement in, an aspect which greatly affects the developer is compilation time. If you have to take pauses all the time to wait for your code to compile it disturbs your work flow. Therefore CM utilizes a runtime environment and an incremental just-in-time compilation, cutting compilation time down to in most cases a matter of seconds. Another thing that slows programmers down is to restart the program you currently use and then recreate a certain situation or find a certain dialog before being able to test how changes in the code behave. CM allows the developer to load new code during execution without restarting the application.

CM also introduces the file as a syntactic unit. This allows for file-private functions only accessible from within the file and also static variables defined in a file and accessible from all classes and functions in the file. This allows some very creative programming (see example 2.1). CM also allows new syntax to be defined through macro definitions. The for-loop, for example, is not a native language construct but based on the while-loop.
Example 2.1: A CM example

```java
public int counter = 0;

public class Test {
    public constructor(str message) {
        pln(message);
        counter ++;
        pln(#counter);
    }
}

{  
    Test("Hello World!");  
}
```

When run this little piece of code will result in the output:
Hello World!
counter=1

The next time it will produce the output:
Hello World!
counter=2

Since the value of the global variable counter is stored by the runtime environment it will be incremented every time a new class is instantiated.

2.1.2 C with self developed pre-processor

Configura AB has a sizable code base of old C code. They use a self developed set of macros to implement a crude variant of object orientation. Every object answers to the same calls but can answer them in different ways. Functions are loosely coupled with their class and the same function can be a part of several classes. The system is old and unintuitive in that, for example, you can call on a function in another class to operate on a class value resulting in faulty execution. There is also a notion of package inheritance. Instead of packages using or importing one another, packages extend each other and thus form a package hierarchy similar to the class hierarchy. To resolve what classes are referenced, `#include` statements
are added to the top of the file including other files. When going through the
include statements the first entity found that matches is the one that is used.
This means that changing the order of file-includes in any file could change the
execution. In Example 2.2 there is a short example of a class definition. First two
methods are defined using class2 as a keyword. The class WorkChair1 is defined
with the same keyword thus linking together the class and the methods.

--- Example 2.2: A C example ---

```c
local VOID defmethod(class2, reshape)(SNAPPER obj)
{
    mReshape(WorkChair1, obj, mSurfaceRGBColor(obj, ChairSeat),
             mSurfaceRGBColor(obj, ChairBack));
}

local VOID defmethod(class2, drop)(SNAPPER obj, POINT p)
{
    super(drop)(obj, p);
    StdCommonSetStandOnObj3Dz(obj, p);
}

mBuildSnapper(WorkChair1, class2, inherits=Chair)
{
    mSetWorkChairSurfaces(class2, hasArms=false);
    mBuild(WorkChair1, snapRadius=0.8, args=(nil,
          mSrfStateRGBColor(MatteBlue),
          mSrfStateRGBColor(MatteBlue)));
}

mBuildSelectIcon(WorkChair1, layer=2);
```

2.1.3 Development environment: Emacs

Configura uses Emacs as development environment. Emacs is a very stripped
environment compared to more modern integrated development environments but
allows great code focus and an uncluttered view. Text manipulation is also very
fast and convenient in Emacs if you’re used to it and know all the shortcuts, but
Emacs has steep learning curve. Configura has added extra features to Emacs
in the form of scripted functions allowing the CM-programmer to jump between
functions and find where functions and classes are defined. These features rely
on the active CM runtime environment which interacts with Emacs. There is
also the possibility to see a text representation of a class hierarchy, but this is
not considered very helpful since the representation as pure text limits the visual
appeal.

2.1.4 Handwritten Documentation

At Configura there is also some handwritten documentation concerning the language, such as a formal clause grammar specifying how the language works and information about the compiler. There is also a forum and a wiki where personnel have added content that discusses the language and the programs.

2.2 Expectations and demands

When designing a documentation system there are many aspects to think about and people in different positions in the company have very diverse opinions. To get a sense of what the expectations on the system were, casual interviews were held with around eight people in varying positions within the company.

2.2.1 Management

The driving force behind this project is the operative manager, who wants to be able to present an attractive package to entice third party developers to work with their platform. They want CM to be able to compete with other commercial languages. During the interview they made a lot of references to solutions like Javadoc and seemed to want something along those lines. Something that makes it easier to learn the system when you are still learning. At present all new personnel are educated internally and mostly learn by experimenting and asking older employee, yet this is not viewed as a problem and seems to be working fine. If new personnel at the company profit from the system it is considered a bonus.

2.2.2 Project managers and developers

Managers of the different departments however see a lot of ways of using documentation for training new recruits and make it easier for personnel to move between departments and between platforms. Although developers feel the development environment is efficient and that navigating source code in Emacs is fine, some confess that more documentation would have been appreciated when they were new at the company. Some employers also expressed an interest in call-graphs, profiling or other more advanced tools to help with debugging and inspection.

2.2.3 R&D

At the research and development department there were many different views on the whole matter of documentation. While some considered documentation to be something completely unnecessary, others where enthusiastic and had a plethora of ideas.

The main issues for those who had a rather negative view on documentation are the problems with keeping the documentation up to date. They general opinion
was that reading the source code is the easiest and most reliable way to gain
knowledge about the code base.

Others had a lot of amazing ideas for what documentation could be like. Most
ideas were outside the scope of the project but things to consider for the future.
Ideas like including audio files in the documentation. If integrated with the de vel-
opment environment, programmers could explain things verbally while writing the
code which would make it easier and allocate less time for documentation. Another
idea was that since the applications that Configura works with are graphical ap-
plications, many objects have a graphical representation which could be generated
and included in the documentation.

Some features were mentioned by almost everybody, including class hierarchies,
information about classes and their methods. Call graphs and package dependan-
cies also seemed of general interest. A call graph shows how different parts of a
program call functions and methods in the program. Package dependencies shows
which packages that use one another.
Chapter 3

Problem description

In this chapter I will present the problems in focus. First the general problem and the questions raised by these problems will be described, then I will review some existing systems and present some research already performed in the field.

3.1 General problem description

To put it loosely, the problem is one of getting from A to B where A is an existing sizable code base and B is some kind of visualization of the software. The problem is information extraction and formatting as well as visualizations and presentation.

Extracting information from existing code bases can be done in several different ways. We can immediately disregard the notion of using predefined tags the way Javadoc does since the code already exists and doesn’t contain any tags. Feasible ways would include getting information from the compiler. The drawback here is that the compiler in most cases completely disregards comments in the code. In addition, compilers often loose some semantic meaning by not keeping track of where code belongs. File names, directory names, white space, the order of functions and that of methods within a class all adds to the semantics making the code much more readable for programmers. One way to solve this would be to write a compiler which is aware of the structure of the code and the program. One other way to extract information would be to write a parser and parse the source code. Depending on what kind of information we intend to extract, the parser could make a less-than-complete parse of the code. In the simplest case you could simply parse the document for class declarations and then have a list of all classes in the code base. If you start to look for more complex information this approach is somewhat similar to the light-weight lexical search proposed by Murphy and Notkin[23] where they use context dependent patterns to extract information.
Problem description

3.2 Relevant information to extract

Deciding which types of information to extract is naturally dependent on the language from which you extract and what kind of analysis you want to perform. If you want something similar to Javadoc, then names of classes and their members and the class structure would be relevant while information about which functions that access a certain variable is unnecessary. If you, on the other hand, would like to be able to construct call graphs or data dependencies graphs, that kind of information would become necessary. There is also information the programmer added to make the code more understandable, such as comments and whitespace [11]. Preferably you would extract as much information as possible and then selectively analyze and display information relevant to the current situation. There are however two limitations to be taken into account. The first is performance. The more information and the more advanced information extracted, the longer will the extraction take. The second is that the method of extraction limits what kind of information you could extract. You could separate the information provided by source code into different categories. The first distinction would be between explicit and implicit information. Implicit information is somewhat difficult to pin down (see figure 3.1).

3.2.1 Explicit semantic information

In this category we have names of classes and parameters and function and so on. Obviously the name of a class should contain semantic clues to the function of the
class. This information is easy to access and easy to present to the user. In the simplest case you could do a pattern search for class definitions and the extract the class name.

### 3.2.2 Implicit semantic information

Implicit semantic information could, for example, be the amount of whitespace between lines in the code and between elements.

### 3.2.3 Explicit structural information

Sub class and super class relationships between classes is typical structural information. Information about which methods and variables that belong to which class is another example. This kind of information can be extracted from the source code without a lot of hassle, though one has to take into consideration that there might be several classes with the same name. The structure of the actual program could be placed in this category as well. This includes function calls, variable accesses and the general flow of the program. To extract this you would have to parse or interpret the code, taking things like scope and reference resolution into consideration.

### 3.2.4 Implicit structural information

The order of which methods are specified within a class is an example of implicit structural information. Other examples are the file in which a function is specified as well as file names and their placement within directories. This information is easy to extract but harder to make any explicit analysis of. It will vary greatly between different languages. In some languages there is a strict one class per file policy while in other languages you could put several related classes in the same file. In the later example the fact that the classes are placed in the same file is a very strong indicator of some sort of semantic connection between the classes.

### 3.2.5 Macros

Macros can hold any kind of semantic and structural information possible. When a macro is interpreted or compiled it is transformed into new source code and some of its semantic context is lost. If possible it is best to access macros in their original form and perform the analysis there. Since documentation and code analysis is for the benefit of the programmer, you need to analyze and document what the programmer actually can see which is the uncompiled macro. Determining the function of an uncompiled macro is a complex task and it is highly domain dependent what kind of macros there is.

### 3.3 Extraction methods

There are several ways to extract information from source code.
3.3.1 Parser

Writing a parser is in one sense a pretty straightforward procedure. The parser traverses the document and collects the relevant information on the way. However, the complexity of the task varies with the complexity of the information to be extracted. Extracting names of classes and methods is simple. Trying to piece together which superclass a class is subclassing from name alone is a bit trickier. Then you have to resolve the entity reference. If there are several classes with the same name you have to determine which one is referenced by looking at scope and at imported files and imported packages. Trying to piece together some kind of call graph requires a lot of reference resolution and, depending on the size of the code base, can slow down the system noticeably. Depending on how you write your parser you can make it possible for it to handle code that’s only half written or broken and doesn’t compile. This is a definite advantage compared to using output from compilers which by definition requires input that is compilable.

Another approach is to use a parser generated from the grammar of the language [16]. This approach has been successfully implemented for both for Java [20] and C++ [9].

3.3.2 Compilation output

Since the compiler parses the code there is a possibility to take advantage of that and use the information collected by the compiler. However most compilers discard their internal data and do not make it available. Sometimes you can bypass this and there are for example ways to hack into g++ and get at the intermediate representation. But this representation can be cumbersome to work with, though some compilers are starting to allow access to compilation information in a more ordered way. For example, IBM Visual Age C++ stores all information gathered during compilation and allows access via an API. This product has been used to create a C++ fact extractor which extracts to cppML [20].

Compilation output mainly consists of an AST (Abstract Syntax Tree) which is a tree structure representing all syntactic information in the source code. Every node of the tree is an element in the language. As always with compilers the main issue is that comments and other semantic information are not necessary included since the compiler has no use for it.

3.3.3 Querying an accessible Compiler

If you are fortunate enough to work with a compiler which you can query more directly about the source code under compilation, then the extraction becomes simplistic. This of course requires that the compiler keeps track of the things you are interested in. Working with the compiler directly can lead to the same problems as working with compiler output since compilers often disregard whitespace and comments. However having a compiler you actually can ask about variable accesses and function calls makes the process of making call graphs and data dependency graphs a lot faster then with a parser.
3.4 Previous research

There has been some substantial work in the field. Documentation and reverse engineering are vast and very diverse fields.

Documentation and especially the term 'good documentation' is hard to specify. Documentation could be anything written about a program from user manual to inline documentation in source code. For this thesis I will concentrate on documentation intended for people who work with developing and maintaining a system. Documentation is a much debated issue in software development and there are many opinions but surprisingly little evidence supporting any specific view of documentation. There are two major views of documentation. The traditional one that stipulates that documentation is necessary and vital to any software development project. Both inline code documentation and external documentation such as records of design decisions are considered as vital. Then there are the followers of extreme programming and agile principles that regard documentation as unnecessary work and propose that close working teams, continual customer contacts and loose hierarchies can minimize the need for any documentation, though extreme programming still approves of some inline documentation embedded in the source code [5].

None of these views have any substantial evidence to support them and since extreme programming still is in its infancy there is no experience with legacy system developed according to this methodology.

The most prominent critique of documentation and the reason it is not widely used even when it exists is the problem of keeping the documentation updated and relevant. Documentation is essentially outdated as you write it. For this reason inline documentation has gained in popularity in the last decades since it is easier to update documentation regarding the portion of the code that you are changing if you can immediately spot it. Inline documentation, however, do not record any structural information about the program as a whole and requires the user to know what they are looking for and browse the source code. To solve the first problem, there are a lot of ancillary documentation extractors, e.g. Pydoc which extracts the comments from the source code and represents them in an appealing form such as a webpages. The second problem is not really a problem since the program on its own contains all structural information needed if you can just get it. Programs which extract structural information from source code are usually categorized as reverse engineering\(^1\).

Many of the later attempts to solve these problems tend to use XML in different ways. Reasons for using XML are that it is widely recognized as standard and has gained a large interest from the open source community as well as from the academic world and that it supports multiple query languages and complex transformations [18]. Transformations can be used to refactor code [10] or to transform data into visual applications (See figure 3.2)[16].

\(^1\)Reverse engineering has two related meanings: reverse engineering from binary code or reverse engineering in the sense of trying to understand legacy systems where you have access to the source code. Here we are referring to the second meaning of the concept.
3.4.1 Existing formats for code representation

There are several proposed formats for representing source code. Most of them are XML based. Some of these deal with a particular language like JavaML [4, 20], which provides a different representation of Java source code, and cppML [20] for C++, while others are more general, like OOML (Object Oriented Markup Language) [20]. In general these languages have an important thing in common which is that they often work from compilation output and have little support for comments and other implicit data. SrcML [19, 8] however takes another approach and simply adds xml tags to source code without making any other changes, and in this way keeps comments and white space. These different XML applications represent information on different levels of abstraction. Al-Ekram made a framework using a combination of language specific and more general languages to make general analysis on different languages (See figure 3.3)[2].

3.5 Existing systems and solutions

There are a many existing solutions principally divided in the two categories of ancillary documentation generation and reverse engineering.

3.5.1 Reverse Engineering tool: RiGi

The most developed and quite old reverse engineering tool is RiGi [3, 22]. RiGi analyses source code representing it as a graph (see figure 3.4). RiGi allows the user to look at the system in different views and analyze it. To enhance the possibility to understand the graph representation RiGi allows graph manipulations such as collapsing groups of nodes to form subsystems [26].

3.5.2 Graph visualization tool: SHriMP

SHriMP is an acronym for Simple Hierarchical Multi-Perspective and is a domain-independent visualization technique for complex information spaces [24]. SHriMP shows a nested hierarchical view of graph data (see figure 3.5). SHriMP supports advanced navigation of the graph through different sorts of zooming. There is also the possibility to move and group nodes when trying to make sense of the data. SHriMP is designed to handle different information spaces and can display
Figure 3.3. Suggested framework[2]
Figure 3.4. Example of a RiGi view
any sort of graph data in GXL (Graph eXchange Language). SHriMP can be used to visualize call graphs, for example, but also to display ontologies and other information spaces. SHriMP is developed as a graphical improvement of RiGi [25].

### 3.5.3 Documenting with the help of structured comments

This approach uses specially structured comments entered in the source code by the user to provide meta information about the code. The most well known among these systems is Javadoc [14]. In Javadoc, documentation is extracted from source code and represented as linked web pages (see figure 3.6). Documentation is extracted from comments where information about the code is tagged using special keywords (see example 3.1). The most interesting keyword is @see which provides contextual information regarding which classes are related in some way. A problem here as with all handwritten documentation is that the function can be updated
while forgetting to update the docstring.

Another, more flexible of these systems is for example Doxygen [27] which can handle comments written in a variety of formats and can extract some information even from undocumented code.
Example 3.1: Java Comment

```java
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 * 
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```
Chapter 4

Implementation

In this chapter I will explain in detail how a prototype was implemented to prove the feasibility of the solution. The chapter will also include a brief glossary of techniques used.

4.1 Proposed solution

I propose an extendable platform for documentation similar to Al-Ekram's framework discussed in section 3.4.1 as this will allow for the system to handle the two different languages and their different extraction methods. It will also allow different types of documentation to be generated from the same knowledge base which is known as one source publishing. However the framework is a little bit too general and there is the problem with information loss between levels. Our simplified framework only uses one level of intermediate representations. The implementation will also use the notion of XSLT as a means to create visualizations of the content of an XML document described in section 3.4.

The system is built in modules which makes it easy to extend by adding more modules. Since persons with different positions in the company have very different needs, an extendable approach allows for different views to be implemented to serve the different information needs. Other benefits of a modularized system is how easy it is to integrate new components. By adding new languages and adding new views and ways to present the information the system can evolve to meet new needs within the company. It can cut development time since you can develop core functionality first and then add more specialized views and do more complex analysis at a later date.

Incorporating different languages in the same framework also means a great reuse of code since only an extraction method for the new language needs to be added instead of having to implement everything from scratch. For developers moving between platforms and languages, it will also help that the documentation looks and behaves in a similar manner.
Figure 4.1. Basic structure of solution
4.2 System overview

4.1.1 Source code

As you can see in figure 4.1 the source code is at the bottom and provide all the information that the rest of the system needs. Source code almost always exists of collections of text documents in folders. In this case code written in C and CM, the internally developed language. There are so many different languages but many of them share similarities and can be ordered into different categories like functional or object oriented. CM is object oriented and though C is not object oriented in itself in this case, due to macros, it behaves like an object oriented language.

4.1.2 Intermediate representation

The intermediate representation should be a way to store information about different programming languages in a uniform way. As shown in figure 4.1 the representation works like a bridge between the data and the viewing applications. This allows the components at higher levels to work in a uniform way regardless of the origins of the data. What information and how it is stored has to be carefully considered to obtain a good balance between storing as much informations as is needed without storing so much information as to slow the system down.

4.1.3 Viewing components

The viewing components could be of a number of origins. They could be already available tools or tools developed together with the rest of the system. As shown in figure 4.1, tools work on data from the intermediate representation as doesn’t have access to the source code. Leveraging the power of existing tools is often a matter of selecting the appropriate information from the intermediate data and exporting it to a readable format. For example export XML data to HTML and design a CSS (Cascading Style Sheet) to be able to view your data in a web browser.

4.2 System overview

To prove the feasibility of the solution a number of modules were implemented and integrated with one another to form an system. Two different language extraction modules and several different modules for displaying the information was developed (see figure 4.2).

4.2.1 Intermediate XML representation

XML is becoming a standard for representing hierarchical data and most existing formats for expressing formal languages are XML based (see section 3.4). Apart from its hierarchical capabilities, XML has a number of general advantages.

- Human Readable
  Most analyses are automated and users have no real need to see the representation. However having an intermediate format which is readable aids the development of tools and integration with third party software.
Figure 4.2. System layout
- **Extendable**

  Since the needs of programmers change and languages evolve, the representation should be easy to extend. XML makes it easy to extend or add to a format and most XML applications are robust in the sense that if you add further information to your representation, then programs written with the old representation in mind will still work unless you made major changes.

- **Widely supported**

  XML is widely supported and there exist numerous applications dealing with displaying and analyzing XML, such as XML Notepad[7]. Since a prerequisite was that the solution would be easy to integrate with third party solutions this is a heavily weighing advantage.

As noted before in section 3.4.1, there exist defined XML applications for expressing information about source code. The ones that could be interesting to use in our application are either SrML, OOML or FactML as they are language independent. SrML is in some ways very verbose and a little bit more than we need. OOML and FactML are both too language independent to express things that are unique to our languages. The simple choice and the approach chosen for this project is to design an XML application able to express the data that we are interested in. One strength with XML is that it is so easy to transform one format into other formats, so long as they contain similar information. In this way information structured in our own format could easily be converted into OOML with a moderate loss of information. The information lost is the information that is specific and can not be expressed in a generic way.

**XML**

XML an acronym for eXtensible Markup Language and it has been developed by W3C (the World Wide Web Consortium). XML is derived from SGML (Standard Generalized Markup Language). SGML was developed in 1986 as a standardized way to add meta-data to documents. This is accomplished by adding "tags" to the document. In other derivations of SGML like HTML you only have a fixed number of tags to use while XML allows the user to define tags most appropriate to the information in question. A collection of user-defined tags designed to model a specific domain of information is called an XML application. This is not to be confused with an application that uses XML.

In Example 4.1 `<class>` is the opening tag for the class element. It is accompanied by a corresponding closing tag `</class>`. Elements can have attributes, such as an id attribute, uniquely identifying the element. Between the open and close tag there can be both text and other nested elements called children of the element. The class has one child which is the element name. XML has a strict structure, and a document which adheres to these rules is said to be well formed. Another concept that is interesting in XML is the notion of validity. Even though XML in general allows any elements, you can write a DTD (Document Type Definition) specifying exactly which element are allowed within a document. A DTD
Example 4.1: A XML Example

```xml
<class id="92771">
  <comment>
    Isometric view insert animation. Adds functionality for feature searching, and sizing the rectangle while placing.
  </comment>
  <name>IsometricViewRectInsertAnimation</name>
  <parent idref="3388">InsertAnimation</parent>
  <src>
    <url>
      c:\cm2\cm\abstract\industry\isometricViewRectInsertAnimation.cm
    </url>
    <pos>1878</pos>
  </src>
</class>
```

can also specify whether an element is allowed to have children, which attributes it must have and much more. An XML document which complies with the rules in a DTD is said to be valid with regard to that DTD. Example 4.2 shows a possible DTD for our XML example.

**XSLT**

XSLT is a XML application designed to specify transformations on XML documents. XSLT works by specifying what will happen with the elements that are processed. Templates specify how elements and the child elements of that element are handled. The template can modify the element, exchange it for another or simply add simple text elements. Example 4.3 shows an XSLT stylesheet transforming a class element into a html document containing the name of the class.

XSLT contains tags for if statements and supports XPath, which allows you to jump in the document and to send parameters along which allow you to perform almost every conceivable transformation.

**Representing program facts as XML**

Since XML is hierarchical it is easy to represent program facts as XML elements. There are however still the choices regarding what facts to save and how to organize them. With XML there is always the choice between elements and attributes. There is a performance difference but there are pros and cons with both depending on how you access the data, but as we do not know beforehand how the data will
Example 4.2: A DTD Example

```xml
<!ELEMENT class (comment, name, parent, src)>
<!ATTLIST class id ID #REQUIRED>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT name (#PCDATA) #REQUIRED>
<!ELEMENT parent (#PCDATA)>
<!ATTLIST parent idref IDREF>
<!ELEMENT src (url, pos)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT pos (#CDATA)>
```

Example 4.3: A XSLT Example

XSLT stylesheet:

```xml
<xsl:template match=".">
<html><body>
<xsl:apply-templates/>
</body></html>
</xsl:template>

<xsl:template match="class">
  <span class="type big"><xsl:value-of select="name"/></span><xsl:apply-templates select="parent"/>
</xsl:template>
```

XML file:

```xml
<class id="345">
  <name>WindowNB</name>
</class>
```

Resulting file:

```xml
<html><body>
<span class="type big">WindowNB</span>
</body></html>
```
be accessed by tools in the future optimization becomes impossible. Instead other values become more important, like how XML was intended to be used and the readability of the documents.

The attributes are supposed to represent meta data about an element. But what can be considered metadata? Are id, name, parent and package metadata? It depends on your definition of metadata, but to enhance readability only id was chosen to be represented as an attribute. The id attribute will be used to link objects together and to make elements distinguishable from each other. The important entities in both languages are classes, methods, functions, packages, fields and globals. To add another dimension and keep some implicit information, a file element was included as well to represent the file. This keeps the structure of how the code is structured into different files. CM has the physical file as a syntactic unit and in the legacy language several classes can be defined in the same file. Therefore it is necessary to properly represent this syntactic information.

To get the information a few more structured related elements are clustered together. The method and field element are made child elements of the class element, and functions and globals are child elements of the package element. The class, package and file exist on the same level and references between them are made using idRef and id attributes. This is because it is faster to find classes if they are specified as standalone elements, and also because classes belong to a package but are specified in a file and it can not be a child element of both. The other possible solution to this would be to specify files as children of a package and then classes as children of the file element. The main drawback with this is that it leads to very large units of XML which is a storage problem.

Examples of the XML representation can be found in section A.1 in the appendix.

### 4.2.2 Extraction program

For each language a program was developed to extract information from the source code. The two programs have graphical interfaces (see figure 4.3) associated with them allowing the user to choose which parts of the code base to generate from. When extracting from the Configura platform the choice is simply which folders to include. When extracting from the CET platform a list of packages is available to choose from. Packages already extracted from are grey in the listing. When the extraction is finished the Extractor object is streamed and saved to disk. Next time you want to generate more documentation into the same folder the object will be read into memory and supply details of which documentation is already present and then incorporate the new documentation generated with the already existing documents. The CET extractor can also incrementally update the existing documentation.

**Configura Parser**

As previously mentioned in section 2.1.2 Configura has a sizable code base written in C which relies on macros for its object orientation. A lot of semantic information
4.2 System overview

would be lost if the already preprocessed code or the compilation output were analyzed because the information held by macros would not be recognizable. To maintain as much of the information as possible a parser was developed which parses the code as it is viewed by the programmer.

The parser has the characteristics of a recursive decent first parser. The parser is pretty lightweight and processes the tokens but skips the parts of the code that is not relevant. The parser simply scans through all files in the code base, building an object structure representing the classes, methods, files and packages. When all information is stored references between classes are resolved. As previously mentioned, reference resolution is a bit tricky on the Configura platform. Simply put, you loop through the representations of all the files and look at the references made in those files. If the references do not reference something defined in the same file, the files included are searched in order to see if the reference is defined in any of those. There are also special statements like `importClass` that imports a class from a package in a more direct way.

As of now the parser cannot extract as detailed information as the CM Extractor since the information is not as easily accessible. The success of the parser was limited as well by my unfamiliarity with the language and its construct since the language is old and much different from object oriented languages of today. It could probably be improved by somebody with greater understanding of the language, but the parser clearly shows that this approach is possible.

CM Extractor

The extractor written to handle the CM language uses the fact that CM has a runtime compiler that has access to information about all currently loaded code. This way the compiler can be questioned and the desired information easily retried. Even though the compiler keeps track of a lot of information regarding the code, it disregards some of the semantic information which we are interested in, such as comments to the code and the order in which things are defined in files.

Some information, like the order of methods has the potential to be restored by comparing positions in the source code. Even if the compiler does not care about that kind of information, it is still possible to reconstruct with the right methods.

Comments are a bit more tricky since the compiler completely disregards them. One solution to this is to make the compiler aware of comments, and this possibility was discussed with the writer of the compiler and the persons responsible for its development. They were actually positive to this approach but as it is a big change it has to be thought over and researched. The biggest concern is that it could affect the performance and effectiveness of the compiler. The other approach which was used instead was to parse separately for the comments. To find the interface comment to a function the compiler’s source code reference is used and from there it is easy to parse backwards to find the comment preceding the function. This slows the extractor some since parsing is more taxing than simply questioning the compiler.
Figure 4.3. Extraction tool for the different code bases
4.2 System overview

4.2.3 XSLT-processor

The XSLT-processor uses different XSLT stylesheets to transform our own XML into other data formats like HTML and GXL. The implementation uses libxslt [28] which is a C library for XSLT transformations.

4.2.4 XSLT stylesheets

XSLT stylesheets describe transformations to be performed on XML data by an XSLT processor. The implementation utilizes XSLT transformations to transform our XML into a number of different formats, both XML based and other text based formats. Stylesheets is also used as filters to sort out specific data.

Using XSLT transformations for data analysis and implementation makes the implementation easy to extend since all you have to do to add a different view or support for a new format is to write a new stylesheet. Some of the stylesheets used are pretty straightforward and only encompass about 50 lines. Other, more complicated transformations use several stylesheets of hundreds of lines.

Some of the more basic transformations only work on one file at a time, such as transforming a class element into an HTML page. More complicated transformations operate on several different elements stored in different files. XSLT allows jumping between files using file names. Since our files are named to the id of the element stored within it, it is easy to write stylesheets that processes several files.

4.2.5 SVG

SVG is another XML application. It is designed to represent the domain of vector graphics and can therefore be used to visually represent data. The implementation uses Graphviz [1] for graph layout. Graphviz uses a file format called dot which is a graph description format, but since XML is easy to morph into virtually any format that is not a big problem. An XSLT transformation is applied, transforming XML into dot-format.

Graphviz then produces a graph in SVG. The resulting graph can then be used as a part of an online documentation since SVG can be viewed in a web browser using a plugin. SVG describes vector graphics, which means that the resulting graph can be scaled and zoomed without any loss of resolution. Both Firefox and Internet Explorer have plugins for SVG support. At present they are useable but have some problems with not implementing the SVG standard in the same way. SVG can also be viewed in standalone SVG viewers. Figure 4.5 shows a SVG graph as a part of a web page.

There is currently support for SVG graphs for class hierarchy and package dependancies, but SVG can be used to make more advanced visualizations [21].

4.2.6 HTML

HTML is not formally an XML application even though they share great similarities since they both are derivatives of SGML(Standard Generalized Markup
FIGURE 4.4. A view of the simple HTML documentation.
Figure 4.5. A view of the simple HTML documentation
XHTML is a more strict version of HTML and is formally an XML application.

The HTML pages generated by the prototype conform to XHTML. The HTML pages generated show information about classes, packages and files. They also show graphs on package dependancies and class hierarchies. In figure 4.4 you can see an example of an HTML page describing a class named DeleteEntitiesAnimation, and in figure 4.5 you can see the inheritance graph for the same class. The pages are linked with cross references allowing the user to quickly find the pages describing a return type or the parent of a class.

The pages are styled with a cascading style sheet (CSS). The stylesheet is designed to make the documentation look similar to how the code looks in the code editor by coloring the code. This for example makes it possible to more quickly navigate and find the name and parameters of a function, as you are used to using the color coding of source code to quickly find the right place in the code.

The HTML Documentation exists like the XML documentation, as a collection of documents in a directory. A dialog (see figure 4.6) allows HTML documentation to be incrementally generated from an XML archive meaning that documentation for the whole archive does not have to be generated at once, but rather that the HTML archive can be extended and updated until it finally contains all data in the XML archive. There are also ways to get pages with indices to only a portion of the complete HTML information. For example if only a selected package is included in the index that makes browsing and finding classes much faster and easier. Right now it is only possible to have one index at a time for every HTML archive, but this is easy to change.

### 4.2.7 GXL and SHriMP

GXL (Graph eXchange Language) is an XML application. It is developed to store general graph data. Transformation from the intermediate XML format to a graph in GXL is trivial using XSLT. SHriMP is a graph viewer with GXL support. SHriMP is very well suited to our needs since it is tailored to display hierarchical data. To conveniently display our information, a number of different views can be specified to give the user a starting point to begin browsing the graph. Coloring of nodes is controlled as well with, for example, private, package and public methods colored in different shades of the same color. With SHriMP you can view both package dependancies, call graphs and class inheritance (see figure 4.7).

### 4.2.8 Interactive class browser

As a proof of concept a very limited interactive browser was implemented in CM. This browser handles both platforms uniformly and displays a tree view of the class structure of the loaded XML archive (see figure 4.8). This allows the user to find in which file a class is specified and open that file in Emacs. Even though this application is written in CM, it uses none of its own knowledge about the classes it handles. All information displayed is gathered from the XML Archive and parsed by the program. Since CM has support for opening new windows in
Figure 4.6. Dialogs for creating GXL graphs and HTML pages
Figure 4.7. Callgraph on the classes contained in a package.
Emacs and handling buffers, it is easy to integrate with Configura’s development environment.

4.3 Scalability

Examples of reverse engineering systems often deal with data from comparable small systems of a few hundred classes. Therefore there is very little information about how to handle scalability. Trying to visualize a system containing several thousand classes is problematic. In addition to the rendering problem there is also the task of handling the information. When the size of the code base increases so does the information about the system.

An example of this is generating XML from an AST. How do you solve the problem of the resulting file being too large to store on disk? Or too cumbersome to access and process? As the size of the code base increases so does the time needed to extract the necessary information. What do you do when the extraction time becomes longer than people are willing to wait?

Figure 4.8. Class Browser
4.3.1 Storing XML

When storing large amounts of XML there are essentially two choices. Either you could store the XML fragments in an XML-database or you can store it in plain text files.

XML databases

XML databases come in two different varieties: XML interfaces to relational databases (XML Enabled Databases) and Native XML databases. XML interfaces to traditional XML databases take XML as input and convert it to tables and when queries are made to the database the result is converted to XML.

A Native XML Database defines a logical model for XML including at least elements, attributes and PCDATA and has XML documents as its fundamental units instead of table rows. Storage for a Native XML Database can be solved in a number of different ways, such as plain text files, binary data or using another type of database as back end. The database can be queried using XPath and XQuery [15].

Many XML databases are geared towards web development and have interfaces for PHP, SOAP and other web frameworks. There are several open source projects that are promising. The most promising for integrating in this kind of project are Sedna [12] or Xindice [13] that expose APIs for Java and C, but at this time they are not ready for use in this kind of project. This problem sadly disqualified XML databases from use as data storage in this project.

Plain text files

A definite benefit of using plain text files when developing is the readability, which makes it easy to debug the system and makes it fairly transparent which in turn will make it easier to develop new tools and integrate the system with other tools and systems. As XML databases are not feasible at this time it was decided that plain text would be used for this project.

The generated XML code for a program is too large to fit into a file buffer and is therefore stored in several different files. Indexation is handled implicitly by the filesystem based on the filenames of the storage files. For example if you would like to access an XML fragment with the ID 1139 you would read the file 1139.xml. All classes, packages and files have their own XML representation in a dedicated file while methods and globals are stored together with the class or package with which they are associated.

XSLT can access other files than the one in which it starts the transformations and in this way complex transformations can combine the content of several files.

To summarize, storing XML in text files satisfies the needs of our project but is slower due to the large number of disk accesses.
4.3 Scalability

4.3.2 Implementing incrementality

The time required to extract information is dependent on two factors: The amount of source code and how complicated and time consuming the extraction method is. One approach to deal with this is to implement an incremental extraction method. This way you could do a time consuming first extraction and then, when there are changes in the code base, extract information about those changes and incorporate them with the already extracted information instead of having to extract everything again.

Incrementality on two levels

Incrementality has two different sides to it, namely that information does not have to be complete to start with and that more information can be added in time and that information can be updated without recreating all the information.

Propagating change

On thing to take into consideration when implementing incremental functionality is propagating changes. If one class is changed, does that affect other classes? What information needs to be re-extracted. This is completely dependent on the language you are working with and the information you extract. For a normal, object oriented language most changes to a class do not propagate to affect other classes. The two things that could cause propagating change are scope and inheritance and overloading.

For example if you use a package that has a class with the same name as a class in your own package all references to that class name in your package would point to your class. If you then deleted your class all those references will point to the class with the same name in the imported package.

With inheritance and overloading there are similar problems. If you reference a method in a class the reference will point to the overloaded method closest in the object hierarchy to your object. If that method is deleted the reference will now point to a method higher up in the object hierarchy.

In CM, package is not a level of scope thus eliminating the first problem. The second problem is trickier to solve and I could not find a solution to this which did not create too much overhead. Solving the problem of the resulting overhead might have switched the focus from the main topic of the project and would have meant far more work than there was time for in the project’s timeframe. In a live implementation of the system this problem could have been solved by using a change log. In using a change log, a more exact set of affected classes could be calculated and the overhead would shrink markedly.

For the old legacy language, it was difficult to try and isolate the propagating change, and since the older code base is a much smaller one and the parser is relatively fast this was not a highly prioritized problem.
Incremental update

When the parts of the code that have changed are isolated and new information about them generated there is the issue of how to find where that information goes in the information base. For classes with the same name as other classes there is as well a need to distinguish which of the information in the information base references the class.

One way to accomplish this is to use some kind of unique id. For example the language assigns a unique compiler-id to every class to track them. However this id is not consistent and the ids need to be the same in sequential compilations. To accomplish this an id generator was built that assigned an unique id to every entity in the language. In between compilations the generator was streamed to disk as a part of the information archive. When more information needed to be added the generator would be read back into memory and the right ids and references found.

4.4 Modularity

The system is built in a highly modular way with components on different levels interacting with each other. This makes it easy to develop new components and fit them into the current structure.

Since the extraction of information from the language is a separate component from the visualization components, adding support for another language simply amounts to writing a new component that extracts information from that language into the common format. Then all the visualization components can immediately display and analyze that information without any changes being made in their execution.

Adding new analysis or new visualizations is simple as well and only requires that either a new component is written that can work directly on the intermediate representation or that an interface is written that transforms the intermediate representation into a form which an already existing tool can read.
Chapter 5

Conclusion

In this chapter I will present some conclusions and thoughts on what was accomplished as well as some prospective future work.

5.1 Conclusions

The aim of this thesis was to design a system that would extract information from an existing code base and visualize it. Such a design was presented and a prototype was implemented. The prototype is not in any way ready to be used in a business environment but it shows that the proposed design is feasible. The design presents a way to extract information and then visualize it with several different tools. The system is flexible and split into different modules making the information flow in the system easy to follow and makes it easy to implement new modules, adding more complicated features. The system addresses the problem with documentation that is out-of-date. Since the documentation is incrementally updated the updates are fast and can easily be performed often.

One thing that was problematic was the fact that there was no research proving which information is useful. However the modularity of the system means that it is easily adapted to changing needs. Which information and visualizations that are useful might also be a matter of personal taste and depend on which information a certain user easily holds in his or her head. Building the system in a modular way makes it possible for these individuals to choose the visualizations and information that they like. This is an idea that I wanted to develop more but time was limited. One way this idea could have been developed is via bookmarks which could be added to the documentation, enabling users to create favorite sets of information. This way you could personalize your documentation system and have the information you look at the most close at hand.

There is still much to be done in the field of information extraction on different levels. As previously discussed, implicit information seemed to be a very interesting notion. However we could not utilize it to its full potential in this project as it mostly deals with information on a more detailed level then we handled.
5.2 Future work

An important task for the future would be to evaluate the system to find out how users respond to it, if the information is useful and how it can be improved. More research into what kind of information that is useful and how it can be extracted would also be a welcome contribution. In particular, research on how the implicit information that programmers leave behind might be interpreted.

Something could perhaps be done for the internal representation of data to boost performance. The simplest solution would be to link the system to an XML database. Both Sedna and Xindice have released new functionality since the start of this project and may now be useable. Also, with more resources a team of developers could tailor a representation based on XML databases which would speed up the execution time.

Other things to focus on could be integration with RiGi and other visualization tools and integrating support for additional languages.
Bibliography


Appendix A

Code examples

A.1 XML example

Here we will show an example of what information is extracted from a CM program and how it is represented in the intermediate XML format.

A.1.1 Source code

The source code in our example is a method from a class. The original code looks like this:

```java
/**
 * Return a unique Id for a Package.
 */
final public str uniqueVdmsymbol pkgN {
    if mpkg in packageVd3ictionaryN {
        return packageVd3ictionaryogetmpkgNr;
    } else {
        str id s idtounterotojmNr
        idtounter99r
        packageVd3ictionaryoputmpkg# idNr
        name3ictionaryoputmid# pkgNr
        entity3ictionaryoputmid# pkgNr
        return idr;
    }
}
```

The following function will take a function object representing the function above, extract information from it and transfer that information to XML. The function uses an XMLSourceWriter which is a wrapper for a FileWriter that adds XML tags to the information as it writes it to the output file.
/**
 * Write a method to writer.
 */

final private void writeMethod(XmlSourceWriter writer, Function f) {
    writer.startMethod(uniqueId(f));
    writer.addComment(comment(f.src));
    writer.addVisibilityLevel(f.export.exportName);
    writer.addType(f.returnType.toS, uniqueId(f.returnType));
    writer.addName(f.name.toS);
    writeFormalArgList(writer, f.formalArgs);

    if (f.isMethod()) {
        if (f.Method.isAbstract) {
            writer.startAttributeList();
            writer.addAttribute("abstract");
            writer.endAttributeList();
        }
    }

    writeFunctionRefers(writer, f);
    writer.endFunction();
}

Here we can see the resulting XML. Every XML fragment is assigned a unique id, which in this case is 7969. These ids are then used to reference other fragments that can be classes or functions or other entities in the language. For example this method takes a CetExtractor as an argument and refers to CetExtractor with its id which is 994. Information about formal arguments and which functions and variables that are accessed are represented.

<method id="7969">
  <comment>
    Return a unique Id for a Package.
  </comment>
  <visibilityLevel>public</visibilityLevel>
  <type idref="909">str</type>
  <name>uniqueId</name>
  <formalArgList>
    <formalArg>
      <type idref="994">CetExtractor</type>
      <name>this</name>
    </formalArg>
    <formalArg>
      <type idref="1020">symbol</type>
      <name>pkg</name>
    </formalArg>
  </formalArgList>
</method>
A.2 XSLT examples

Since line breaks matter for the result of the execution, line breaks which are only added to enable the code to fit on paper will be marked with an arrow ->.

A.2.1 GXL (Graph eXchange Language) example

Here we will show how a part of a call-graph is generated. The following parts of a the stylesheet exportToGxl.xsl was applied to the XML generated in section A.1.

```xml
<!--
   - Add a node representing the method and add an edge from the
   - class containing it.
   -->
<xsl:template match="method">
  <xsl:param name="id"/>
  <node id="{@id}">
    <xsl:text disable-output-escaping="yes">&lt;type xlink:href="" ->
    <xsl:text disable-output-escaping="yes"></type&gt;
    <xsl:value-of select="visibilityLevel"/>
    <xsl:value-of select="name"/>
    <xsl:value-of select="contains"/>
    <xsl:if test="type/@idref">
      <xsl:if test="" returns"="" ->
      <xsl:text disable-output-escaping="yes">&lt;type xlink:href="returns"/>
      <xsl:text disable-output-escaping="yes"/>
    </xsl:text>
  </node>
</xsl:template>
</xsl:stylesheet>
```
<xsl:apply-templates select="functionCall">
  <xsl:with-param name="id" select="@id"/>
</xsl:apply-templates>
<xsl:apply-templates select="variableAccess">
  <xsl:with-param name="id" select="@id"/>
</xsl:apply-templates>
<xsl:apply-templates select="createsClass">
  <xsl:with-param name="id" select="@id"/>
</xsl:apply-templates>
</xsl:template>

<!--
- Add an edge from the function to the other function being called.
-->
<xsl:template match="functionCall">
  <xsl:param name="id"/>
  <xsl:if test="@idref">
    <edge from="{$id}" to="@idref"><xsl:text ->
    -> disable-output-escaping="yes">&lt;type xlink:href="calls"/ &gt;</xsl:text></edge>
  </xsl:if>
</xsl:template>

<!--
- Add an edge from the function to the variable being accessed.
-->
<xsl:template match="variableAccess">
  <xsl:param name="id"/>
  <xsl:if test="@idref">
    <edge from="{$id}" to="@idref"><xsl:text ->
    -> disable-output-escaping="yes">&lt;type xlink:href="accesses"/ &gt;</xsl:text></edge>
  </xsl:if>
</xsl:template>

To create the following GXL:

<node id="7608">
  <type xlink:href="public+Method"/>
  <attr name="name"><string>uniqueId</string></attr>
</node>
<edge from="902" to="7608"><type xlink:href="contains"/></edge>
<edge from="7608" to="925"><type xlink:href="returns"/></edge>
<edge from="7608" to="7728"><type xlink:href="calls"/></edge>
<edge from="7608" to="7730"><type xlink:href="calls"/></edge>
<edge from="7608" to="1881"><type xlink:href="calls"/></edge>
<edge from="7608" to="7491"><type xlink:href="calls"/></edge>
<edge from="7608" to="7733"><type xlink:href="calls"/></edge>
<edge from="7608" to="7734"><type xlink:href="calls"/></edge>
<edge from="7608" to="7735"><type xlink:href="calls"/></edge>
<edge from="7608" to="1402"><type xlink:href="calls"/></edge>
<edge from="7608" to="6520"><type xlink:href="calls"/></edge>
<edge from="7608" to="7588"><type xlink:href="accesses"/></edge>
<edge from="7608" to="7589"><type xlink:href="accesses"/></edge>
<edge from="7608" to="7630"><type xlink:href="accesses"/></edge>
<edge from="7608" to="7580"><type xlink:href="accesses"/></edge>

A.2.2 HTML example

Here we will show a segment of the HTML documentation. The following parts of the stylesheet exportToHtml.xsl were applied to the XML generated in section A.1.

<!--
   - Transforms an method.
  -->
<xsl:template match="method">
  <xsl:apply-templates select="comment"/>
  <xsl:apply-templates select="visibilityLevel"/>
  <xsl:text> </xsl:text>
  <xsl:apply-templates select="type"/>
  <xsl:apply-templates select="name"/>
  <xsl:apply-templates select="formalArgList"/>
  <xsl:apply-templates select="attributeList"/>
  <br/>
</xsl:template>

<xsl:template match="comment">
  <span class="comment"/>
</xsl:template>

<!--
   - Transform a visibilityLevel element.
  -->
<xsl:template match="visibilityLevel">
  <span class="visibilityLevel"></xsl:apply-templates/>
</xsl:template>
<!--
   - Transforms the type element in a formal argument list.
   -->
<xsl:template match="formalArg/type">
   <span class="type">
      <a>
         <xsl:attribute name="href"><xsl:value-of select="@idref"/>&gt;
            &gt; .htm</xsl:attribute>
         <xsl:apply-templates/>
      </a>
   </span>
</xsl:template>

<!--
   - Transforms a type element.
   -->
<xsl:template match="type">
   <span class="type">
      <a>
         <xsl:attribute name="href"><xsl:value-of select="@idref"/>&gt;
            &gt; .htm</xsl:attribute>
         <xsl:text> </xsl:text><xsl:apply-templates/>
      </a>
   </span>
</xsl:template>

<!--
   - Transform a generic name.
   -->
<xsl:template match="name">
   <span class="name"> <xsl:text> </xsl:text><xsl:apply-templates/></span>
</xsl:template>

<!--
   - Transform a formal argument list.
   -->
<xsl:template match="formalArgList">
   ( <xsl:apply-templates/>)
</xsl:template>
A.2 XSLT examples

<!--
- Transform a formal argument.
-->
<xsl:template match="formalArg">
  <xsl:choose>
    <xsl:when test="position()=1">
    </xsl:when>
    <xsl:when test="position()=2">
      <xsl:apply-templates/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>, </xsl:text><xsl:apply-templates/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--
- Transform a defaultValue.
-->
<xsl:template match="defaultValue">
  =<span class="defaultValue">xsl:value-of select="."</span></xsl:template>

... 

To create the following HTML:

<span class="comment">
  /**<br/>
  * Return a unique Id for a Package.<br/>
  */<br/>
</span>

<span class="visibilityLevel">public</span> <span class="type">-></span> <span class="name">-></span> <a href="909.htm">str</a><span class="name">-></span> <span class="name">uniqueId</span>(<span class="type">-></span><a href="1020.htm">symbol</a><span class="name">-></span> pkg)</n
<span class="name"></span>

Which looks like this viewed in a browser when styled with a stylesheet that adds color coding:

/**
* Return a unique Id for a Package.
*/
public str uniqueId (symbol pkg)
Appendix B

DTD for intermediate representation

This is the DTD that regulates how the XML documents might be structured.

```
intermediateRepresentation.dtd
<!ELEMENT package (name, uses*, function*)>
<!ATTLIST package id ID #REQUIRED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT uses (#PCDATA)>
<!ATTLIST uses idref IDREF #REQUIRED>

<!ELEMENT function (comment?, visibilityLevel, type, name, formalArgList, (functionCall | variableAccess)*)>
<!ATTLIST function id ID #REQUIRED>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT visibilityLevel (#PCDATA)>

<!ELEMENT type (#PCDATA)>
<ATTLIST type idref IDREF #REQUIRED>

<!ELEMENT formalArgList (formalArg*)>
<!ELEMENT formalArg (type, name)>

<!ELEMENT functionCall (#PCDATA)>
<!ATTLIST functionCall idref IDREF #REQUIRED>
```
<!ELEMENT variableAccess (#PCDATA)>
<!ATTLIST variableAccess idref IDREF #REQUIRED>

<!ELEMENT createsClass (#PCDATA)>
<!ATTLIST createsClass idref IDREF #REQUIRED>

<!ELEMENT class (comment?, name, parent?, packageRef, attributeList?, src, child*, constructor*, field*, method*)>
<!ATTLIST class id ID #REQUIRED>

<!ELEMENT attributeList (attribute+)>  
<!ELEMENT attribute (key, value?)>  
<!ELEMENT key (#PCDATA)>  
<!ELEMENT value (#PCDATA)>  

<!ELEMENT parent (#PCDATA)>
<!ATTLIST parent idref IDREF #REQUIRED>

<!ELEMENT packageRef (#PCDATA)>
<!ATTLIST packageRef idref IDREF #REQUIRED>

<!ELEMENT src (url, pos)>  
<!ELEMENT url (#PCDATA)>  
<!ELEMENT pos (#PCDATA)>  

<!ELEMENT child (#PCDATA)>
<!ATTLIST child idref IDREF #REQUIRED>

<!ELEMENT field (comment?, visibilityLevel, type, name, attributeList?)>  
<!ATTLIST field id ID #REQUIRED>

<!ELEMENT constructor (comment?, visibilityLevel, type, name, formalArgList)>  
<!ATTLIST constructor id ID #REQUIRED>

<!ELEMENT method (comment?, visibilityLevel, type, name, formalArgList, attributeList?, (functionCall | variableAccess)*)>  
<!ATTLIST method id ID #REQUIRED>

<!ELEMENT file (name, packageRef, contains*)>  
<!ATTLIST file id ID #REQUIRED>

<!ELEMENT contains (#PCDATA)>  
<!ATTLIST contains idref IDREF #REQUIRED>
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Åsa Wihlborg]
Using an XML-driven approach to create tools for program understanding