Evolution Doctor: A Framework to Control the Evolution of Undocumented Software Systems

Massimiliano Di Penta

Università degli Studi del Sannio
Dipartimento di Ingegneria
Tesi di Dottorato di Ricerca
in Ingegneria dell’Informazione - XV Ciclo

Advisor: Prof. Giuliano Antoniol
TABLE OF CONTENTS

LIST OF TABLES ................................................................. v
LIST OF FIGURES ................................................................. viii
LIST OF APPENDICES ........................................................... xii
LIST OF ACRONYMS .............................................................. xiii
PREFACE ............................................................................. xv

CHAPTER

I. Introduction ................................................................. 1

Bibliography ................................................................. 15

II. Analyzing an Predicting Software System Evolution .... 18

2.1 Introduction ................................................................. 18
2.2 Related Work ............................................................... 19
   2.2.1 Analyzing and Predicting Software Evolution ........ 19
   2.2.2 Analyzing and Predicting Clones ......................... 20
2.3 Background Notions ...................................................... 21
   2.3.1 Introduction to Time Series ............................ 21
   2.3.2 Clone Detection Process ............................... 27
2.4 Predicting Evolution via Time Series ......................... 33
   2.4.1 Metrics Extraction .................................... 34
   2.4.2 Model Building ......................................... 34
   2.4.3 Predicting Evolution and Model Validation ....... 34
2.5 Case Studies ............................................................... 35
   2.5.1 The Linux Kernel ..................................... 35
   2.5.2 mSQL .................................................. 38
2.6 Predicting the Evolution: Results ............................. 40
2.7 Observing the Cloning Evolution of the Linux Kernel ...... 46
   2.7.1 Kernel 2.4.18 Analysis ........................... 49
   2.7.2 Analyzing Cloning Evolution ....................... 53
2.7.3 Answers to Questions on Cloning Evolution .... 55
2.8 Predicting Cloning Evolution .............................. 56
  2.8.1 Case Study Results .................................... 57
2.9 Conclusions .................................................. 59
Bibliography ..................................................... 62

III. Analysis of Multi-Configuration Code ...................... 68
  3.1 Introduction .............................................. 68
  3.2 Preprocessor-Conditioned Types .......................... 69
    3.2.1 Examples of Faults .................................. 70
  3.3 Related Work ............................................ 71
  3.4 Architecture of Type-Checking Tool .................... 72
    3.4.1 Building the Enhanced Symbol Table ............... 74
    3.4.2 Composing and Simplifying Expressions ........... 77
    3.4.3 Type-Checking ....................................... 78
  3.5 Impact of Preprocessor Conditionals ................... 79
  3.6 Conclusions ............................................. 82
Bibliography ..................................................... 83

IV. Design Pattern Extraction .................................... 85
  4.1 Introduction .............................................. 85
  4.2 Related Work ............................................ 87
  4.3 Design Patterns ......................................... 88
  4.4 Recovery Process ....................................... 91
    4.4.1 Design Pattern Library ............................ 92
    4.4.2 AOL Representation and Parsing ................... 95
    4.4.3 Class Metrics Extraction ......................... 95
  4.5 Pattern Recognition ..................................... 96
    4.5.1 Class Level Metrics Constraints .................. 99
    4.5.2 Shortest Path Constraints ......................... 101
    4.5.3 Structural Constraints ........................... 102
    4.5.4 Delegation Constraints ............................ 104
  4.6 Tool Support ............................................ 105
  4.7 Case Studies ............................................ 106
    4.7.1 Case Study Description ............................ 106
    4.7.2 Case Study Results ................................ 108
  4.8 Conclusions ............................................. 112
Bibliography ..................................................... 115

V. Source Code Directory Structure Reorganization ............ 119
  5.1 Introduction ............................................. 119
5.2 Related Work ........................................... 120
5.3 Background Notions: a Primer on Concept Analysis ........ 121
5.4 The Method .............................................. 122
  5.4.1 Executables Identification and Modules Dependences Detection ..... 124
  5.4.2 Directory Organization ................................ 124
  5.4.3 Tool Support ....................................... 127
5.5 Case Studies ............................................. 128
  5.5.1 Case Study Description ............................... 128
  5.5.2 Case Study Results ................................ 129
5.6 Conclusions ............................................ 131
Bibliography .................................................. 133

VI. Library Identification and Miniaturization ....................... 136

6.1 Introduction ........................................... 136
6.2 Related Work ........................................... 139
6.3 Background Notions .................................... 141
  6.3.1 Clustering ....................................... 141
  6.3.2 Genetic Algorithms ................................. 143
6.4 The Library Reorganization Process ........................ 145
  6.4.1 Software System Graph Representation ............... 145
  6.4.2 Graph Construction ................................ 147
  6.4.3 Handling Unused Objects ............................ 148
  6.4.4 Removal of Circular Dependencies among Libraries 148
  6.4.5 Identification of Duplicates Symbols and Clones .... 150
  6.4.6 Library Miniaturization ............................. 151
  6.4.7 Determining the Sub-Optima Libraries by Hierarchical Clustering .... 153
  6.4.8 Identification of new Libraries ...................... 159
  6.4.9 Library Remodularization using Dynamic Information .... 159
6.5 Tool Support ............................................ 163
  6.5.1 Architecture for Collecting Dynamic Information .... 164
6.6 Applying the Miniaturization Process: Case Study ........... 166
  6.6.1 GRASS ........................................... 167
  6.6.2 Handling Unused Objects ............................ 168
  6.6.3 Removal of Circular Dependencies among Libraries 169
  6.6.4 Identification of Clones ........................... 169
  6.6.5 Library Miniaturization ............................. 172
  6.6.6 Extraction of new Libraries ........................ 176
6.7 Library Identification with CA: Case Studies ................. 176
6.8 Dynamic Miniaturization: Case Studies ....................... 179
  6.8.1 Case Study Description ............................. 180
  6.8.2 Case Study Results ................................ 181
LIST OF TABLES

Table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Lehman’s laws.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Linux Kernel’s most important events.</td>
<td>36</td>
</tr>
<tr>
<td>2.2</td>
<td>CRs $\geq$ 1% among major subsystems.</td>
<td>51</td>
</tr>
<tr>
<td>2.3</td>
<td>CRs $\geq$ 10% among mm architecture dependent code.</td>
<td>52</td>
</tr>
<tr>
<td>2.4</td>
<td>CRs $\geq$ 5% among drivers.</td>
<td>52</td>
</tr>
<tr>
<td>2.5</td>
<td>Actual TS Values MRE (mSQL 1.0.x).</td>
<td>57</td>
</tr>
<tr>
<td>2.6</td>
<td>Actual TS Values and MRE (mSQL $\beta$ releases).</td>
<td>57</td>
</tr>
<tr>
<td>2.7</td>
<td>Actual TS Values and MRE (mSQL 2.0.x).</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>An example of enhanced symbol table.</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>Public-domain code characteristics.</td>
<td>107</td>
</tr>
<tr>
<td>4.2</td>
<td>Industrial code characteristics.</td>
<td>108</td>
</tr>
<tr>
<td>4.3</td>
<td>Reduction of candidates through the stage filters for the Adapter design pattern.</td>
<td>109</td>
</tr>
<tr>
<td>4.4</td>
<td>Results of pattern instances recovery on public-domain code.</td>
<td>110</td>
</tr>
<tr>
<td>4.5</td>
<td>Results of pattern instances recovery on industrial design.</td>
<td>110</td>
</tr>
<tr>
<td>4.6</td>
<td>Structural design pattern recovery on ET++.</td>
<td>112</td>
</tr>
<tr>
<td>5.1</td>
<td>Objects needed to build each executable (including library files).</td>
<td>123</td>
</tr>
<tr>
<td>5.2</td>
<td>Objects needed to build each executable (without library files).</td>
<td>124</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.3</td>
<td>Sample example: concepts.</td>
<td>125</td>
</tr>
<tr>
<td>5.4</td>
<td>Sample example: concepts after pruning common attributes.</td>
<td>126</td>
</tr>
<tr>
<td>5.5</td>
<td>Samba module dependencies.</td>
<td>130</td>
</tr>
<tr>
<td>5.6</td>
<td>MySQL module dependencies.</td>
<td>131</td>
</tr>
<tr>
<td>6.1</td>
<td>Objects needed to build each executable (including library files).</td>
<td>151</td>
</tr>
<tr>
<td>6.2</td>
<td>GRASS key characteristics.</td>
<td>167</td>
</tr>
<tr>
<td>6.3</td>
<td>Results of clone detection.</td>
<td>170</td>
</tr>
<tr>
<td>6.4</td>
<td>GRASS largest libraries.</td>
<td>172</td>
</tr>
<tr>
<td>6.5</td>
<td>Results of the library miniaturization process before considering feedback (w_3 = 0).</td>
<td>173</td>
</tr>
<tr>
<td>6.6</td>
<td>Results of the second round of the library miniaturization process (w_3 \neq 0).</td>
<td>175</td>
</tr>
<tr>
<td>6.7</td>
<td>Performance comparison between pure GA and hybrid GA with hill climbing.</td>
<td>176</td>
</tr>
<tr>
<td>6.8</td>
<td>MySQL 2.0.8 retrieved libraries vs. real libraries (.o files).</td>
<td>177</td>
</tr>
<tr>
<td>6.9</td>
<td>Samba 2.0.5 retrieved libraries vs. real libraries (.o files).</td>
<td>179</td>
</tr>
<tr>
<td>6.10</td>
<td>Dynamic miniaturization: case studies characteristics.</td>
<td>180</td>
</tr>
<tr>
<td>6.11</td>
<td>Results for (\lambda = 1).</td>
<td>183</td>
</tr>
<tr>
<td>6.12</td>
<td>Results for (\lambda = 0.5).</td>
<td>183</td>
</tr>
<tr>
<td>7.1</td>
<td>ALBERGATE (\lambda_k' = 0) traceability results.</td>
<td>218</td>
</tr>
<tr>
<td>7.2</td>
<td>ALBERGATE single words plus bigrams traceability recovery results.</td>
<td>220</td>
</tr>
<tr>
<td>7.3</td>
<td>Transient Meter (\lambda_k' = 0) baseline results.</td>
<td>220</td>
</tr>
<tr>
<td>7.4</td>
<td>Transient Meter: (\lambda_k' = 0) results after removal of automatic-generated identifiers.</td>
<td>221</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.5</td>
<td>Transient Meter: $\lambda'_k = 0$ final results (after removal of non-traceable classes).</td>
<td>222</td>
</tr>
<tr>
<td>7.6</td>
<td>Transient Meter: single words plus bigrams traceability results.</td>
<td>223</td>
</tr>
<tr>
<td>7.7</td>
<td>Library Management: $\lambda'_k = 0$ results after morphological analysis.</td>
<td>223</td>
</tr>
<tr>
<td>7.8</td>
<td>Library Management: $\lambda'_k = 0$ results after removing automatic-generated identifiers.</td>
<td>223</td>
</tr>
<tr>
<td>7.9</td>
<td>Library Management: $\lambda'_k = 0$ results after removing non-traceable files.</td>
<td>224</td>
</tr>
<tr>
<td>7.10</td>
<td>Library Management: single words plus bigrams traceability results.</td>
<td>224</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Evolution Doctor framework activities.</td>
<td>11</td>
</tr>
<tr>
<td>1.2</td>
<td>The Evolution Doctor framework.</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>Block diagram of ARMA model.</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>The clone identification process.</td>
<td>28</td>
</tr>
<tr>
<td>2.3</td>
<td>Linux Kernel stable and experimental releases evolution.</td>
<td>38</td>
</tr>
<tr>
<td>2.4</td>
<td>mSQL KLOCs and procedure number evolution.</td>
<td>39</td>
</tr>
<tr>
<td>2.5</td>
<td>Linux Kernel evolution in terms of KLOCs: timeline releases.</td>
<td>41</td>
</tr>
<tr>
<td>2.6</td>
<td>Linux Kernel evolution in terms of cyclomatic complexity: timeline releases.</td>
<td>41</td>
</tr>
<tr>
<td>2.7</td>
<td>Linux Kernel evolution in terms of # of functions: stable releases.</td>
<td>42</td>
</tr>
<tr>
<td>2.8</td>
<td>Linux Kernel evolution in terms of cyclomatic complexity: stable releases.</td>
<td>43</td>
</tr>
<tr>
<td>2.9</td>
<td>Timeline releases - MRE on KLOCs prediction.</td>
<td>43</td>
</tr>
<tr>
<td>2.10</td>
<td>Timeline releases - MRE on cyclomatic complexity prediction.</td>
<td>44</td>
</tr>
<tr>
<td>2.11</td>
<td>Stable releases - MRE on # of functions prediction.</td>
<td>45</td>
</tr>
<tr>
<td>2.12</td>
<td>Stable releases - MRE on cyclomatic complexity prediction.</td>
<td>45</td>
</tr>
<tr>
<td>2.13</td>
<td>Timeline releases - metrics prediction MMREs.</td>
<td>47</td>
</tr>
<tr>
<td>2.14</td>
<td>Stable releases - metrics prediction MMREs.</td>
<td>47</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Design example with method call information for delegation-based stage.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Main blocks of the process.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Directory organization (dotted lines are intended as symbolic links).</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Hierarchical clustering dendrogram and cut-point.</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Pseudo-code of a GA.</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Example of System Graph.</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Examples of dependencies among libraries.</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Activity diagram of the library miniaturization process.</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Genetic operators.</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>The dynamic miniaturization process.</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>a) Pipe architecture of the trace compressor. b) Deployment diagram of the trace collector system.</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>Splitting library libortho.</td>
<td></td>
</tr>
<tr>
<td>6.10</td>
<td>Silhouette statistics for different number of clusters.</td>
<td></td>
</tr>
<tr>
<td>6.11</td>
<td>New libdbmi layering structure.</td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>Samba 2.0.5 objects lattice.</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>How the programmer can map concepts into code.</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>The traceability recovery process.</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>An example of traceability links.</td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>AST object model example: method declarator.</td>
<td></td>
</tr>
<tr>
<td>A.2</td>
<td>Extracting nodes from a tree.</td>
<td></td>
</tr>
<tr>
<td>A.3</td>
<td>Matching subtrees.</td>
<td></td>
</tr>
</tbody>
</table>
A.4 Computing metrics. ................................. 246
A.5 Example of OCL expression parse tree. .................. 249
A.6 OCL interpreter package diagram. ......................... 252
A.7 Example of OCL interpreter use. .......................... 253
LIST OF APPENDICES

Appendix

A. OCL Based AST Navigator ........................................ 236
LIST OF ACRONYMS

AIC . . . . Aikake Information Criterion
AOL . . . . Abstract Object Language
API . . . . Application Programming Interface
AR . . . . Auto Regressive
ARIMA . . Auto Regressive Integrated Moving Average
ARMA . . . Auto Regressive Moving Average
AST . . . . Abstract Syntax Tree
CA . . . . Concept Analysis
CASE . . . Computer Aided Software Engineering
CORBA . . Common Object Request Broker Architecture
COTS . . . Commercial Off The Shelf
CR . . . . . Cloning Ratio
CR(DMD) . Coverage Ratio of DMD
CR(DMU) . Coverage Ratio of DMU
CVS . . . . Concurrent Versions System
DF . . . . Dependency Factor
DMD . . . . Dynamic Matrix of Dependencies
DMS . . . . Design Maintenance Systems
DMU . . . . Dynamic Matrix of Uses
FF . . . . Feedback Factor
FM . . . . Feedback Matrix
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPE</td>
<td>Final Prediction Error</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographical Information System</td>
</tr>
<tr>
<td>GRASS</td>
<td>Geographic Resources Analysis Support System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Desktop Environment</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IR</td>
<td>Information Retrieval</td>
</tr>
<tr>
<td>LOCs</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>MA</td>
<td>Moving Average</td>
</tr>
<tr>
<td>MD</td>
<td>Matrix of Dependencies</td>
</tr>
<tr>
<td>MMRE</td>
<td>Mean Magnitude Relative Error</td>
</tr>
<tr>
<td>MRE</td>
<td>Magnitude of Relative Error</td>
</tr>
<tr>
<td>mini SQL</td>
<td>mini SQL</td>
</tr>
<tr>
<td>MU</td>
<td>Matrix of Uses</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMT</td>
<td>Object Modeling Technique</td>
</tr>
<tr>
<td>OO</td>
<td>Object-Oriented</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistants</td>
</tr>
<tr>
<td>PPR</td>
<td>Probabilistic Partitioning Ratio</td>
</tr>
<tr>
<td>PR</td>
<td>Partitioning Ratio</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>RLE</td>
<td>Run Length Encoding</td>
</tr>
<tr>
<td>SF</td>
<td>Standard Deviation Factor</td>
</tr>
<tr>
<td>SG</td>
<td>System Graph</td>
</tr>
<tr>
<td>SMB</td>
<td>Service Message Block</td>
</tr>
</tbody>
</table>
SMD . . . . Static Matrix of Dependencies
SMU . . . . Static Matrix of Uses
SOAP . . . . Simple Object Access Protocol
SQL . . . . . Structured Query Language
SRS . . . . . Software Requirements Specifications
TS . . . . . Time Series
UML . . . . Unified Modeling Language
XML . . . . Extensible Markup Language

PREFACE

Real world software systems undergo to repeated maintenance activities during their lifetime. It has been estimated that from 40 to 90 percent of software life-cycle costs are related to maintenance. Due to the market pressure and to the need for having back the system operational in the shortest time possible, these maintenance activities tend to deteriorate the underlying software structure and documentation. Some examples of negative side-effects of maintenance interventions are the growth of cloning percentage, the increase of library size, the presence of unused objects, the lost of source file organization and traceability links.

This thesis proposes a framework, named Evolution Doctor, to diagnose and cure such phenomena. First and foremost, the framework allows to analyze and predict several aspects of software system evolution (such as size, complexity, cloning). Second, the framework defines a set of methods and tools to diagnose and cure the problems: reorganize libraries, restructure the source file directory organizations, identify design patterns into object-oriented code, and recover traceability links.
The framework activities can be classified as both reverse engineering activities, aiming at extracting high-level representations from source code, and reengineering activities, exploiting reverse and forward engineering activities to re-vitalize the structure of the software itself. The main target of such activities is to significantly reduce maintenance costs, as well as to improve software system understandability.
CHAPTER I

Introduction

Any software system undergoes, during its lifetime, to change interventions for maintenance and evolution purposes. Software maintenance can be defined as “the modification of a software product after its delivery to correct faults, improve performance or other attributes, or to adapt the product to a changed environment” [1].

As stated in the IEEE Standard for Software Maintenance [2], software maintenance interventions may be classified into:

- **Corrective maintenance**, i.e., the reactive modification of a software product performed after delivery to correct discovered faults;

- **Adaptive maintenance**, i.e., the modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment;

- **Perfective maintenance**, i.e., the modification of a software product after delivery to improve performance or maintainability; and

- **Emergency maintenance**, i.e., unscheduled corrective maintenance performed to keep a system operational.

Software evolution can be due to several, different possible reasons. First and foremost, software systems use tends to elicit new requirements. In fact, when a system is operational the user can realize what he/she can potentially do with it;
moreover, new requirements may be due to the environment change, or to new organizational needs. Finally, the market pressure and the introduction of new technologies push activities such as the porting of the software system on new hardware/software platforms, the integration with new devices, or the adoption of new user interfaces.

Seminal works on studying software evolution were performed by Lehman [3, 4, 5, 6], who studied a number of software systems over the years. Lehman defined two types of software systems: the S-type systems and the E-type systems. An S-type system is a system where the criterion of success is that the program satisfies its specification. In other words, this means that, to prevent ambiguity, program properties should be formally Specified (S stays for Specified). S-type programs are the bricks for construction of E-type systems (E stays for Evolutionary). E-type systems model the real world, or at least a part of it:

- The acceptability of the system is determined by its behavior and by the results of its execution;
- Satisfaction of stakeholders’ current needs is the ultimate criterion of validity;
- Needs, desires, opportunities, operational domain evolve; and, therefore
- Intrinsic need for evolving the application itself.

Given this, Lehman formulated the eight laws of software evolution for E-type systems, reported in Table 1.1.

As strongly suggested by the standard [2] and by several studies, software maintenance and evolution activities should follow precise phases, in order to keep all the documentation (i.e., all the deliverables of the different life cycle phases) correctly and consistently aligned, and also to keep high the quality of the source code produced/changed. In particular, the IEEE standard [2] suggests that a maintenance process should encompass the following phases:

- Problem/modification identification, classification and prioritization;
- Analysis;
<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Continuing Change</td>
<td>An E-type system must be continually adapted else it becomes progressively less satisfactory in use</td>
</tr>
<tr>
<td>II</td>
<td>Increasing Complexity</td>
<td>As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it</td>
</tr>
<tr>
<td>III</td>
<td>Self Regulation</td>
<td>Global E-Type system evolution processes are self-regulating</td>
</tr>
<tr>
<td>IV</td>
<td>Conservation of Organizational Stability</td>
<td>Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime</td>
</tr>
<tr>
<td>V</td>
<td>Conservation of Familiarity</td>
<td>In general, the average incremental growth (growth rate trend) of E-type systems tends to decline</td>
</tr>
<tr>
<td>VI</td>
<td>Continuing Growth</td>
<td>The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime</td>
</tr>
<tr>
<td>VII</td>
<td>Declining Quality</td>
<td>Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type system will appear to decline as it is involved</td>
</tr>
<tr>
<td>VIII</td>
<td>Feedback System (Recognized 1971, formulated 1996)</td>
<td>E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems</td>
</tr>
</tbody>
</table>

Table 1.1: Lehman’s laws.

- Design;
- Implementation;
- Regression/system testing;
- Acceptance testing and
- Delivery.

In practice, the above process (or any similar one) is rarely adopted, due to the market pressure, the necessity of emergency maintenance interventions (almost never followed, as suggested by the standard, by documentation alignment and source code refactoring), or the limited resources or budget available. As a consequence, the maintainability of the system and, in general, its quality, tends to deteriorate. Documentation, executables, sub-systems, modules and files/directories organization, the as-is architecture, may no longer correspond to the original software architecture. In this scenario program comprehension and software maintenance activities are extremely difficult: any change may produce unpredictable side effects on other portions of the system. Moreover, repeated undocumented maintenance interventions on the code reduce one’s understanding of the system, substituting reliable information on the software with smoky perceptions and folklore [7].
This makes desirable some forms of software system reengineering and redocumentation. Reengineering will not only revitalize the system, but will also provide reusable material for future systems [2]. While reengineering techniques have always been made available for traditional engineering disciplines, software reengineering is a relatively new field.

Reengineering is generally composed of two steps: a reverse engineering step and a forward engineering step. The reverse engineering step provides a view of a software system at different levels of abstraction: this means redocumenting code with diagrams, extracting properties (e.g., identifying design patterns, clones, traceability links), and also performing measurements, identifying problems, bottlenecks (e.g., big libraries, circular dependencies). The forward engineering step is devoted to enhance, in different ways and with different goals, a software system. Reengineering differs from restructuring, that is the transformation from one representation to another at the same level of abstraction (e.g., transforming non-structured to structured code).

The term “reverse engineering” comes from the analysis of hardware, with the objective of extracting design from a finished product. Hardware reverse engineering is regularly applied in military environments as well as in any competitive market to analyze a competitor/adversary’s product.

The purpose of software reverse engineering is to obtain a sufficient level of understanding to aid maintenance as well as new development. As detailed in [8], the objectives of software reverse engineering can be summarized in six points:

1. Cope with the complexity of a software system, extracting relevant information;

2. Generate alternate views highlighting different aspects of the system itself;

3. Help to detect anomalies and side effects;

4. Synthesize higher level abstractions; and

5. Facilitate reuse, in that high-level views of the system can help to detect candidate reusable assets from present systems.
Reverse engineering techniques can produce significantly improvements in the cost of software. It is widely recognized that software maintenance represents from 40 to 90 percent of the total life-cycle costs [9, 10]. Given the increment of productivity reverse engineering can produce in the maintenance phase, but also across the entire software life-cycle, the total system expense can be greatly reduced.

This thesis aims to propose a framework, named *Evolution Doctor*, able to monitor software system quality, giving to the maintainer some tools to help redocument/reorganize the system itself. The *Evolution Doctor* proposes and puts together reverse engineering techniques aiming at extracting high-level representations of the software system, with forward engineering techniques exploiting such representations to perform different kind of improvements on the software system itself. In particular, the framework encompasses the following monitoring activities:

- **Analysis and prediction of software evolution**: before planning any kind of reorganization/redocumentation intervention, project managers should analyze the software evolution trends. It is in fact widely recognized [11] that these operations should be performed when the evolution trend is stable. In order to analyze/predict software evolution trends, a method based on TS (Time Series) forecasting has been developed. It is worth noting that, besides the identification of stable trends suitable for software refactoring/reorganization, the applications of such approach can be different, from development/maintenance planning to effort/cost estimation;

- **Analysis and prediction of cloning evolution**: when writing a device driver, porting an existing application to a new processor, or for any other reason, developers may decide to copy an entire working subsystem and then modify the code to cope with the new hardware. This technique ensures that their work will not have any unplanned effect on the original piece of code they have just copied. However, this evolving practice promotes the appearing of duplicated code snippets, also said *clones*. The literature proposes various methods for identifying clones in a software system [12, 13, 14] and [15, 16, 17, 18].
There is not so much work in the study of the evolution of similar code fragments among several versions of the same software system [19, 20, 21]. As a software system evolves, new code fragments are added, certain parts deleted, modified or remain unchanged, thus giving rise to an overall evolution difficult to represent by fine-grained similarity measures. The proposed framework therefore identifies, relying on some empirical studies performed on large open source systems, the analysis of cloning evolution as an important indicator of software system refactoring. Not surprisingly, it has been experienced that as software evolves, new clones are introduced but, at the same time, existing ones tend to be refactored;

- **Analysis of configuration-dependent code:** the porting of software systems to different hardware platforms or operating systems, as well as the introduction of drivers for particular devices (e.g., adding new network or soundcard drivers to an operating system kernel) is often carried out producing portions of configuration-dependent code (implemented, in C and C++, using preprocessor conditionals). Indeed, the number of possible configurations a software system can be compiled into tends to reach, especially for some widely adopted open source systems (e.g., Linux) the order of several thousands, many of which are unlikely to be ever instantiated. This may be the cause of several problems. In fact, while commonly used configurations are type-checked by iterating the compilation across them, rarely used configurations are unlikely to be type checked, and in such configurations a variable may have a type not compatible to its use or it may contains uses of variables never defined. To assess the spreading of such kind of problems, the proposed framework encompasses an approach to identify all possible types each variable declared in a software system can assume, and under which conditions, as well as to detect potential inconsistent variable uses;

- **Design patterns extraction:** when monitoring the evolution of OO (Object-Oriented) software systems, it is worth investigating on the presence of design
patterns into design documents and source code. OO design patterns are the
typical mechanism used to organize together classes performing a specific task.
In other words, design patterns represent well-known solutions to common de-
sign problems in a given context. The most well-known OO design patterns
collection is contained in the book of Gamma et al. [22]: 23 design patterns
were collected and documented by the authors who also presented pattern
implementation in Smalltalk and C++. Other design patterns are described
in [23] and, in general, using the design pattern description explained in [22]
any design pattern catalogue can be easily extended adding new patterns. The
framework presented in this thesis proposes an approach for the extraction
of structural design patterns from source code or from design documentation.
While forward engineering the benefit of using design patterns is clear [24],
from a program comprehension and maintenance perspective a pattern pro-
vides knowledge about the role of each class within the pattern, the reason for
certain relationships among pattern constituents and/or the remaining parts
of a system. In other words, the discovery of patterns in a software artifact
highlights rationale of the adopted solution, representing a step in the pro-
gram understanding process and improving documentation. Consequently, in
maintenance, the identification of design pattern instances provides insight on
software artifact structure and reveals places where changes, reuse, or exten-
sions are expected. Moreover, design pattern extraction can give to maintainers
a measure of source code/design quality and, finally, helps in the identification
and extraction of components from existing software systems.

Once being able to monitor the above described aspects of software evolution, it
is possible, when necessary, to apply techniques aiming at improving the structure
of the software system itself. In particular, the reengineering activities proposed by
the Evolution Doctor framework consist of:

- **Source code directory structure reorganization**: an important aspect
  that contributes to software system maintainability is the organization, in
proper directory hierarchies, of its source code files. It should be useful that, instead of having source code files flattened in a single directory, files used by common sets of applications were put together, and that the dependency relationships between applications and source code directories were graphically highlighted. However, such organization sometimes does not exist, it is incomplete or it tends to be deteriorated during maintenance and evolution (i.e., files are added in the wrong position). An approach for source code reorganization highlighting module dependencies has been therefore developed;

- **Library identification and miniaturization:** one of the undesired effects of software evolution is also the proliferation of unused components, or components unlikely to be used by a given subset of software system applications. As a consequence, the size of binaries and libraries tends to grow. At the same time, a major trend of today’s software market is the porting of applications on hand-held devices or, in general, on devices having a limited amount of resources available. Several forms of interventions and, in particular, the miniaturization of libraries and applications, are therefore necessary. The core part of the proposed framework copes with several aspects of software miniaturization, such as removing unused objects and code clones, as well as creating small, cohesive libraries and splitting the existing ones. The last step has been implemented using a hybrid approach based on hierarchical clustering, GA (Genetic Algorithm)s and hill climbing, also incorporating the developer’s knowledge. Most of the miniaturization activities are language independent (relying on object module analysis) thus they do not require any kind of source code parsing and are applicable to software systems developed with different programming languages. The framework also encompasses the possibility of clustering based on dynamic information, keeping into account dependencies exploited during application execution in a given user profile. This allows to effectively cluster objects that, at run time, have an high likelihood to be used together in the same context or by a common set of applications. Such dynamic
information has been extracted instrumenting the code and the collecting execution traces using a distributed software architecture, based on web services. It is worth noting that dynamic information should be properly complemented by static information to also take into account dependency links not exploited at run-time;

- **Traceability links extraction**: the latest aspect that has been considered in this work deals with extracting traceability links between artifacts produced during different software life cycle phases. Traceability links can be helpful for several tasks [25] such as program comprehension, software maintenance and impact analysis. Traceability may be distinguished, according to [26], in *vertical traceability*, i.e., traceability between artifacts produced at different life cycle phases (requirements, design documents, source code, user manuals, test suites, etc.) and *horizontal traceability*, i.e., traceability between artifacts produced at the same phase (e.g., traceability between related requirements, between requirements and use-cases, etc.). Software maintenance and evolution often tend to deteriorate traceability links, that, because of the limited time/effort available, the strict time-to-market, tend to be no longer updated. Stemming from previous proposed approaches [25, 27], the framework encompasses an approach to extract traceability links based on the consistency of use of identifiers in high level documents (e.g., requirements) and in source code: whenever a term (composed of one or more adjacent words) in the requirements is always mapped to the same identifier in the source code, the proposed approach is able to extract traceability links once few links are known. The approach aims also to identify factors, such as automatic generated code, COTS (Commercial Off The Shelf), middlewares that, if not properly used, contributed to negatively affect traceability.

What above stated is summarized in Figure 1.1, where the *Evolution Doctor* framework is split in two main “areas”:

1. A *diagnose area*, aiming at measuring some peculiar properties of the software
system, also analyzing predicting their trends; and

2. A cure area, where the software system underwent activities devoted to its re-vitalization.

As shown in Figure 1.1, the diagnose area should also allow maintainers to simply extract new metrics from software systems, where the analysis supplied by the framework does not suffice or, in general, where there is the need for a deeper analysis. The framework therefore proposes to use the OCL (Object Constraint Language) for expressing queries over an object model representing the AST (Abstract Syntax Tree) of the source code to be analyzed. OCL is part of the UML (Unified Modeling Language) lingua franca and thus several advantages can be readily obtained. Central to the idea is to shift the analysis paradigm from a tree-based to an OO paradigm, and to provide a meta-model decoupling the query language from the target language.

Design patterns extraction can be considered a diagnose activity, in that it can constitute a quality measure of the source code itself. However, it can also be thought of as a cure activity, allowing to extract and factor reusable components.

Figure 1.2 gives a complete overview of the framework, highlighting the artifacts produced at different levels of abstraction and their dependencies.

Analysis starts on artifacts produced at different stages of the software life-cycle. Although the aim of this thesis (as also stated from the title) is the evolution control of undocumented software systems, most of the activities start relying on the more available (and reliable) source of information: the source code. However, some forms of reverse engineering (such as design pattern extraction) can also be performed on design documents. Finally traceability links between artifacts at different levels of abstraction (for instance between source code and design or between source code and requirements) are extracted. Thus traceability link extraction requires that any other form of documentation (to be linked to source code) is available.

Starting from the bottom of the graph, the upper level built on the source code is constituted of AST and the set of object modules obtained compiling the source
### Figure 1.1: Evolution Doctor framework activities.

code itself. As it will be explained in chapters V and VI, object modules will be used to build a software system *Dependency Graph* independently from source code analysis. Of course, the alternative is to build dependency graphs directly from the AST. The framework also extracts different kind of information from the AST, in particular:

- A representation of class diagrams extracted from OO source code, named AOL (Abstract Object Language) (see Chapter IV);

- The weighted list of identifiers (identifier index);

- A symbol table, also accounting for variables declared inside preprocessor conditionals (see Chapter III);
Figure 1.2: The Evolution Doctor framework.
• Analysis of problems derived from preprocessor-dependent code. This kind of analysis, explained in Chapter III, also relies on the conditioned symbol table mentioned above;

• Different kind of source code metrics (e.g. LOCs (Lines of Code), cyclomatic complexity, etc.). As described above and in Appendix A, an OCL based metrics extractor has been conceived and developed; and

• An instrumented version of the source code.

As said, object modules can be used to produce a **Dependency Graph**. Similarly, by compiling instrumented code we will obtain object modules and, after linking, instrumented binaries. The execution of such binaries will produce a set of execution traces, from which it is possible to extract a **Dynamic Dependency Graph**. Dependency Graphs can be used to factor object into libraries (e.g., splitting bigger libraries into smaller ones, see Chapter VI), to recover makefiles and to obtain a source code directory structure particularly suitable to perform maintenance tasks (the last two activities are detailed in Chapter V).

Moving slightly on the left, the figure highlights that metrics are used, as described before, for **clone detection**. Then, predictions about the evolution of the systems, in terms of metrics, but also percentage of clones, can be performed (see Chapter II).

The AOL, extracted from source code (but also from design documents) is used for design patterns extractions, as detailed in Chapter IV. Finally, after indexing both source code identifiers and terms extracted from analysis or design documents, it is possible to extract traceability links (see Chapter VII).

For the development/enhancement of each activity proposed in the framework, an empirical approach has been followed, as also reflected from the structure of the different chapters:

• Each proposed approach is firstly described in detail, after a proper background of related work in-the-field and of mathematical tool used (e.g., clustering, GA) has been given;
• After implementing proper toolkits, the approach is validated performing analyses and reengineering activities on real-world software systems, ranging from over 1 MLOCs open source systems to some industrial software systems. The effectiveness of the approach is finally measured by means of proper indicators (e.g., forecasting error, quality of the clusters produced, etc.).

This thesis is organized as follows. Chapters from II to IV describe the *diagnose* activities, while chapters from V to VII describe the *cure* activities. In particular, Chapter II reports an overview on studies on software evolution, and describes a method to study/predict software evolution trends in terms of some most significant metrics, as well as in terms of clones. Chapter III presents an approach to detect variable inconsistencies in preprocessor-dependent code, while Chapter IV describes a method for the extraction of structural design pattern from source code or from design documentation.

Chapter V describes a method to reorganize source code files and to understand their directory structure. Chapter VI represents the core of this thesis work, proposing the framework activities related to library reorganization and miniaturization. Finally, Chapter VII describes an approach for traceability links extraction and to identify traceability affecting factors. Appendix A describes the architecture of the OCL-based metrics extraction tool.
Bibliography


[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley Publishing Company, Reading, MA, 1995.

http://www.mindview.net/Books/TIPatterns/.


CHAPTER II

Analyzing an Predicting Software System Evolution

2.1 Introduction

As said in the introduction, and as also stated from Lehman’s laws on software evolution [1, 2, 3], it is widely recognized that a program must evolve to meet the user’s ever changing needs. In a competitive market, it is essential to effectively manage software evolution. In particular, cost and effort estimation are important aspects of software project management. Experience shows that accurate estimation is difficult, since an average error of 100% may be considered good and an average error of 32% outstanding [4]. Most methods for effort estimation require an estimate of the size of the software. Once a size estimate is available, models can be used to relate size to effort. Cost estimation is not a one-time activity performed at a project beginning. Estimates should be refined continually throughout a project [5]. Thus, it is necessary to estimate size repeatedly throughout software evolution.

The software evolution is naturally affected by the phenomenon of evolution of duplicated portions of code snippets, also said *clones*. Often, the system is originally conceived as a single platform application, with a limited number of functionalities and supported devices. Then, it evolves adding new functionalities and can ported to new product families: in other words, new devices and/or target platforms can be added. When writing a device driver, porting an existing application to a new processor or, in general, for any other reason, developers may decide to copy an entire working subsystem and then modify the code to cope with the new hardware. This
technique ensures that their work will not have any unplanned effect on the original piece of code they have just copied. When studying software evolution, it is therefore essential to keep into account how cloning percentage evolves across different releases of a software system.

This chapter proposes an approach, based TS, to predict the evolution of software systems. Similarly approaches widely adopted in disciplines such as economy and automatic control, the proposed approach to predict software evolution is based on black box modeling. This means that external characteristics of software systems, such as LOCs, cyclomatic complexity and number of functions, are observed and predicted. The chapter therefore reports results obtained in analyzing the evolution of a large software system, the Linux Kernel. After studying the evolution in terms of external characteristics, cloning evolution is also investigated. Not surprisingly, while across releases new clones tend to be added, existing clones are periodically refactored. Finally, it will be shown how the same approach applied to predict the evolution of a software system in terms of size may be used to predict the evolution of clones.

The chapter is organized as follows. After a review of the state-of-the-art in analyzing and predicting software evolution, and in the clone detection, background notions about TS and metric-based clone detection are reported in Section 2.3. Then, the method for predicting evolution is described in Section 2.4. Case study description and results are reported in Section 2.5 and 2.6 respectively. Section 2.7 reports and discusses the analysis of the cloning evolution on the Linux Kernel, and Section 2.8 reports results obtained from predicting clone evolution on MySQL (mini

\section{2.2 Related Work}

\subsection{2.2.1 Analyzing and Predicting Software Evolution}

Several studies have been performed to recover the architecture and analyze the evolution of large open source software systems, such as the Linux Kernel. In \cite{6} and \cite{7} Bowman et al. recovered the Kernel architecture; further detailed analyses
were executed by Tran et al. in [8] and [9].

The first experience in analyzing the evolution of the Linux Kernel in terms of metrics was done by Godfrey and Quiang Tu in [10]. Successively, Godfrey and Quiang Tu performed a study of the evolution of one of the subsystem of the Linux Kernel (the SCSI subsystem) also in terms of cloning percentage, and they developed a tool to aid software maintainers in understanding how large system have changed over time. Results of this study are summarized in [11].

First applications of TS to predict software characteristics evolution were reported by Yuen in [12, 13, 14], where the evolution of a large operating system (the IBM OS/360) was analyzed. An empirical approach to predict software evolution has been presented by Kemrer and Slaughter in [15], describing a set of methods and techniques developed or adapted to predict the evolution of some different software systems.

Ramil and Lehman [16] described some models that predict effort as a function of a suite of metrics of software evolution, and presented a study to the evolution of the Kernel of a mainframe operating system. Fuentetaja et al. [17] presented some preliminary studies on modeling and predicting the evolution of OS/360 using TS. Finally, first investigations to predict Linux Kernel evolution using TS, restricted to stable releases, has been reported in [18].

### 2.2.2 Analyzing and Predicting Clones

Previous research studied both the detection and the use of clones for widely varying purposes, including program comprehension, documentation, quality evaluation, or system and process restructuring. Some of the techniques used for clone detection are based on a full text view of the source code [19, 20]. Other approaches, such as those pursued by Mayrand et al. [21] and Kontogiannis et al. [22], focus on whole sequence of instructions (BEGIN-END blocks or functions) and allow the detection of similar blocks using metrics. Kontogiannis et al. [22] detect clones also using two further pattern matching techniques, namely dynamic programming matching and statistical matching between abstract code description patterns and source code.
Another clone detection tool, proposed by Baxter et al. in [23], relies on the comparison of subtrees extracted from the AST of a system. A technique based on token comparison and a tool, named CCFinder, to visualize clones in software systems was presented by Kamiya et al. in [24]. Krinke [25] proposed a clone detection approach based on finding similar subgraphs in attributed directed graphs, considering not only the syntactic structure of programs but also the data flow.

Several applications of clone detection have also been investigated: Johnson [20] visualizes redundant substrings to ease the task of comprehending large legacy systems. Mayrand et al. [21], as well as Lagüe et al. [26], documented the cloning phenomenon for evaluating the quality of software systems. Lagüe et al. [26] have also evaluated the benefits, in terms of maintainability of a system, of the detection of cloned methods. Finally, Baxter et al. [23] restructured systems by replacing clones with macros, in order to reduce the quantity of source code and to facilitate maintenance.

A study devoted to analyze the presence of clones in the Linux Kernel was reported in [27]. Successively, it was investigated how clones evolves across subsequent versions [28]. Only few papers have studied the evolution of similar code fragments among several versions of the same software system [29, 30]. As a software system evolves, new code fragments are added, certain parts deleted, modified or remain unchanged, thus giving rise to an overall evolution difficult to represent by fine-grained similarity measures.

2.3 Background Notions

This section will report an overview of the technique used in this chapter to predict software evolution and to detect clones. In particular, basic notions on TS will be summarized, and the metrics-based approach for clone detection will be described.

2.3.1 Introduction to Time Series

A TS is defined as a sequence of observations, $Y_1, Y_2, \ldots, Y_n$ ordered along a single dimension, such as time. Mathematically, a TS may be thought of as a
collection of random variables ordered in time and defined as a set of time points that may be continuous or discrete.

TS data often arise when monitoring industrial processes, business metrics and, in general, the temporal behavior of a phenomenon. A TS can be distinguished from any other kind of sequence data, in that data points collected over time have an internal structure (autocorrelation, trend, seasonal variation) that should be accounted for.

One of the possible objectives in analyzing TS is prediction: given an observed TS, one may want to predict its future values. This kind of analysis is used in many applications, such as economic forecasting, budgetary analysis, stock market analysis, process and quality control, census analysis, and many more.

There are many methods used to model and forecast TS [31, 32, 33, 34]. Some of the most widely adopted are:

- Box-Jenkins ARIMA (Auto Regressive Integrated Moving Average) models;
- Box-Jenkins Multivariate Models;
- Holt-Winters Exponential Smoothing; and
- Multivariate Autoregression.

Different approaches may be chosen in different cases. The smoothing technique, for example, reveals more clearly the underlying trend of a TS, as well as seasonal and cyclic components. Multivariate models (Box-Jenkins and Multivariate Autoregression) apply when each TS observation is a vector of numbers.

In this section we will focus our attention to univariate TS, recorded sequentially over equal time increments. Time is an implicit variable, therefore this kind of TS can also be applied when predicting a sequence of discrete events rather than the exact time an event will take place (e.g., for predicting characteristics of the next two versions of a software system, we are not interested in knowing when a certain version will be released but, given the current value \( v_i \), we should only predict values \( v_{i+1} \) and \( v_{i+2} \)).
TS may be affected by two main phenomena: non-stationarity and seasonality. A TS is non-stationary if its mean and variance changes over time. In other words, a non-stationary TS exhibits trends or periodic fluctuations (seasonality).

More formally, non-stationarity detection can be reduced by identifying two distinct data segments that have significantly different statistic distributions. Several tests can be used to decide whether two distributions are statistically different: Student’s test, F test, chi-square test and Kolmogorov-Smirnov test [35].

For prediction purposes, a non-stationary TS can be transformed into a stationary TS, with one of the following techniques:

- Differencing data: given the series \( Z_t \) a new series \( Y_t = Z_t - Z_{t-1} \) is created;
- If the data contains a trend, it is possible to remove it (fitting a curve), and then work only on the residuals; or
- A non-constant variance can be stabilized taking the logarithm or the square root.

A TS displays seasonality if it exhibits periodic fluctuations: for example, retail sales may exhibit peak close to Christmas. Autocorrelation plots, spectral plots or box plots may often help to detect seasonality.

2.3.1.1 Modeling Univariate Time Series

There are several approaches to modeling TS. One is to decompose the TS into a trend, a seasonal and a residual component. Holt-Winters exponential smoothing [31, 36] is an example of this approach. Another approach, commonly used in scientific and engineering applications, is to analyze TS in the frequency domain. Details can be found in [37, 38, 39].

Finally, a common approach for modeling univariate TS is given by the ARIMA models, which includes as special cases the AR (Auto Regressive), MA (Moving Average) and the ARMA (Auto Regressive Moving Average) models. The first common approach for modeling univariate TS models is the AR model.
Suppose that $Z_t$ is a discrete purely random process (that consists of a sequence of random variables $Z_t$ mutually independent and identically distributed; purely random processes have constant mean and variance) with zero mean and variance $\sigma^2_Z$, then a process $X_t$ is said to be an autoregressive process of order $p$ (AR($p$)) if

$$X_t = \alpha_1 X_{t-1} + \cdots + \alpha_p X_{t-p} + Z_t$$ \hspace{1cm} (2.1)

where $\alpha_i$ are constants, and $Z_t$ represents normally distributed random errors. In other words, an autoregressive model is simply a linear regression of the current value of the series against one or more prior values (i.e., the present is thought of as a linear combination of the past values). AR models can be analyzed with linear least square techniques.

Another approach for modeling univariate TS models is the MA model. Given $Z_t$ a discrete purely random process with zero mean and variance $\sigma^2$, then a process $X_t$ is said to be a moving average process of order $q$ (MA($q$)) if

$$X_t = Z_t + \beta_1 Z_{t-1} + \cdots + \beta_q Z_{t-q}$$ \hspace{1cm} (2.2)

where $\beta_i$ are constants. Once the backward shift operator $B$ is defined as

$$B^j X_t = X_{t-j}$$ \hspace{1cm} (2.3)

a moving average process can be written as

$$X_t = \Theta(B) Z_t$$ \hspace{1cm} (2.4)

where

$$\Theta(B) = 1 + \beta_1 B + \cdots + \beta_q B^q$$ \hspace{1cm} (2.5)

A MA model is, in other words, a linear regression of the current value of the series against the random shocks of $q$ prior values of the series. The base idea of this model is that random shocks of the current value may be propagated to future values. Iterative non-linear fitting procedures need to be used instead of linear least squares.
A more general class of models for TS is obtained by combining MA and AR processes. A mixed autoregressive moving average process containing \( p \) AR terms and \( q \) MA terms is said to be an ARMA process of order \((p, q)\). It is given by:

\[
X_t = \alpha_1 X_{t-1} + \cdots + a_p X_{t-p} + Z_t + \beta_1 Z_{t-1} + \cdots + \beta_q Z_{t-q} \tag{2.6}
\]

where \( X_t \) is the original series and \( Z_t \) is a series of unknown errors which are assumed to follow the normal probability distribution. Using the backward shift operator \( B \), the previous equation may be written in the form:

\[
\Phi(B)X_t = \Theta(B)Z_t \tag{2.7}
\]

where

\[
\Phi(B) = 1 - \alpha_1 B - \cdots - \alpha_p B^p \Theta(B) = 1 + \beta_1 B + \cdots + \beta_q B^q \tag{2.8}
\]

Figure 2.3.1.1 shows the representation of an ARMA model as a filter. The AR and MA parts both contribute to the output. In particular, the AR part accounts for the feedback: previous TS values are fed back, delayed to contribute to the output. The importance of ARMA processes lies in the fact that a stationary TS may often be described by an ARMA model involving fewer parameters than a pure MA or AR process [38]. However, even if stationary TS can be efficiently fitted by an ARMA process [40], most TS are non-stationary.

Box and Jenkins introduced a generalization of ARMA processes to deal with the modeling of non-stationary TS [34]. In particular, if in equation (2.7) \( X_t \) is replaced by \( \nabla d X_t \), it is possible to describe certain types of non-stationary TS. Such a model is called ARIMA (Auto Regressive Integrated Moving Average) because the stationary model is fitted to the differenced data has to be summed or integrated to provide a model for the non-stationary data. Defining:

\[
W_t = \nabla d X_t = (1 - B)^d X_t \tag{2.9}
\]

the general process ARIMA\((p,d,q)\) is of the form:

\[
W_t = \alpha_1 W_{t-1} + \cdots + \alpha_p W_{t-p} + Z_t + \beta_1 Z_{t-1} + \cdots + \beta_q Z_{t-q} \tag{2.10}
\]
Figure 2.1: Block diagram of ARMA model.

More details can be found in [34]. There are three main stages to build a Box-Jenkins TS model:

1. Model Identification;
2. Model Estimation;
3. Model Validation.

2.3.1.2 Box-Jenkins Model Identification

In order to identify an ARIMA model, the following steps are required:

1. Determine if the TS is stationary: if it is, an ARMA model (i.e., $d = 0$) can be used; otherwise, stationarity must be achieved by differencing data (the degree of differentiation identifies the parameter $d$) or fitting a curve;

2. Detect seasonality: the period of seasonality will be additional input to the model estimation software;

3. Identify the order of AR and MA models ($p$ and $q$). In practice, this step is often difficult if performed only with autocorrelation functions. In the last
years, information-based criteria, such as FPE (Final Prediction Error) and AIC (Aikake Information Criterion) [41] have been developed and used. These techniques are implemented by most of statistical software packages.

2.3.1.3 Box-Jenkins Model Estimation

This step involves the estimate of parameters for the Box-Jenkins models (i.e., the $\alpha_i$ and $\beta_i$ coefficients defined in equation (2.8)). This is a non-linear estimation problem, usually left to statistical software packages, applying techniques like non-linear least squares or maximum likelihood estimation [42].

2.3.1.4 Box-Jenkins Model Validation

The residuals (i.e., the differences between predicted and actual values) have to be analyzed to see if the identified model is adequate. Model validation for Box-Jenkins is similar to model validation for non-linear least squares fitting [31, 40, 41]. This means that the error term $Z_i$ is assumed to follow the assumptions for a stable univariate process (i.e., it behaves like random drawings, or like a fixed distribution [38]). If the Box-Jenkins model fits the data, the residuals should satisfy these assumptions.

2.3.2 Clone Detection Process

Clones are defined as code fragments indistinguishable under a given criterion. As below detailed, the work presented in this chapter was performed considering a metrics-based approach [21]: two pieces of code can be considered as clones, with respect to a set of metrics, if they exhibit the same values for all the metrics of the set.

Different granularities may be considered when extracting clone information (e.g., compound statement or function body). In this work, attention has been focused on function definitions. The process defined to study clone evolution, outlined in Figure 2.3.2, relies on the concept of clone clusters, i.e., sets of indistinguishable functions. The process consists of the following, subsequent phases:
1. Handling of preprocessor directives;

2. Function identification;

3. Metrics extraction; and


![Diagram](image)

Figure 2.2: The clone identification process.

### 2.3.2.1 Handling Preprocessor Directives

Parsing programming languages such as C or C++ poses several challenges. Besides the intrinsic programming language peculiarities (e.g., unions, structs, classes, function pointers, etc.), preprocessor directives must be suitably handled. Preprocessor directives are usually managed by a dedicated compiler component, the *preprocessor* (e.g., the GNU *cpp*). Parsing multi-platform code where preprocessing directives are platform-dependent is equivalent to projecting the source code on a given hardware/software configuration.

To obtain information on several platforms, at least two approaches are feasible:
• Preprocess and parse the code sources with different configurations; or

• Adopt a fictitious reference configuration.

Unfortunately, for large size systems such as Linux, the first approach may not be realistic nor feasible. For example, the Linux 2.4.0 kernel contains more than 7000 files, it runs on ten different processors, 400 preprocessor switches drive the actual kernel configuration. Each preprocessor switch can assume three values:

• Y: the code is included into the compiled kernel;

• N (or commented switch): the code is excluded; or

• M: a dynamically loadable module is produced.

Clearly, among the 10 * 400³ possibilities there are many meaningless configurations (e.g., it is very unlikely that a machine has multiple different sound boards). For further details about problems caused by preprocessor-dependent code see Chapter III.

In this work a coarse-grain analysis has been performed by defining a fictitious reference configuration, thus without identifying any specific architecture. This approach is well suited for the identification of function clones among several platform-dependent sub-systems without recompiling the kernel.

The heuristic adopted to handle preprocessor directives is based on the consideration that very often only the then part of an #ifdef is present; moreover, the then branch almost always contains more code than the else branch. Among the 3243 source (.c) files of the 2.4.0 kernel, 2172 contain at least one #ifdef, whereas only 1140 files have an #else preprocessor directive. The actual number of #ifdef is by far larger (22134) than the number of #else (3565), and, in terms of volume (measured in LOCs), the then branch is an order of magnitude larger (about 300 KLOCs versus 20 KLOCs).

Since preprocessor directives, i.e., #ifdef, must be balanced, a parsing of the preprocessor statements can project the source on if branch; the #ifdef conditions
were forced to be true, thus extracting the then branches. The preprocessor elimination step generates sources with removed preprocessor directives, regardless of the hardware/software architecture. Unfortunately, there are few cases where this heuristic produces syntactically wrong C code. Namely, a C scope (i.e., { }) may be opened in the then part of an #ifdef subject to condition EXP, and the scope end ({ }) be located in a different preprocessor statement within the #else part. This also means that the scope is closed by a combination of expressions where EXP is negated. This is the situation found, for example, in the Linux 2.4.0 ultrastor.c SCSI driver. Due to the very low number of such cases (in the 2.4.0 kernel, twenty on about 48000 functions), these were considered pathological situations, detected and signaled for manual intervention.

2.3.2.2 Function Identification

Large C systems are likely to encompass a variety of mixed programming styles, programming patterns, idioms, coding standard and naming conventions. Most noticeably, both the ANSI-C and the old Kernighan & Ritchie style may be present. A tool inspired by island-driven parsing [43] has been implemented to localize and extract function definitions. Once islands (e.g., function bodies or signatures) were identified, the in-between code was scanned, and the function definition extracted by means of a hand-coded parser.

2.3.2.3 Metrics Extraction

Following the approach proposed in [21], the functions extracted as illustrated above were compared on the basis of software metrics accounting for layout, size, control flow, function communication and coupling. In particular, each function was modeled by 54 software metrics:

- The number of passed parameters;
- The number of LOCs;
- The number of statements;
• The cyclomatic complexity;

• The number of used/defined local variables;

• The number used/defined non-local variables;

• Software metrics accounting for the number of arithmetic and logic operators
  (++, -, >=, <, etc.);

• The numbers of function calls;

• The numbers of return/exit points;

• The numbers of structure/pointer access fields;

• The numbers of array accesses; and

• Software metrics accounting for the number of language keywords (e.g., while, if, do).

Different set of metrics could indeed be adopted (e.g., those used in [21]). However, it was experienced that, on sufficiently large systems, the use of different sets of metrics does not significantly influence the results. Function names and file/unit names were not used as metrics.

2.3.2.4 Clone Cluster Identification

Studying commonalities between software systems/sub-systems, function identity may be disregarded in favor of a different concept: clone clusters. A clone cluster can be seen as a set of similar code fragments which contains identical fragments or fragments exhibiting negligible differences from a given fragment prototype. Each pair of functions was compared, and, exact metrics identity was required to classify two functions as clones. This assumption corresponds to the ExactCopy and DistinctName classes presented in [21].

Let $M_f = < m_1(f), \ldots, m_n(f) >$ be the tuple of metrics characterizing a function $f$, where each $m_i(f)$ ($i = 1 \ldots n$) is the $i$-th software metric chosen to describe $f$ (e.g.,
number of passed parameters, number of LOCs, cyclomatic complexity, number of used/defined local variables, and number used/defined non local variables).

For any given function \( f \), let \( C_f \) be the \( f \)th clone cluster. \( C_f \) is the subset of function \( g \) belonging to the considered software system/sub-system \( S_k \), that exhibits software metric values \( m_i(g) \) identical or similar to \( m_i(f) \):

\[
C_f \overset{\text{def}}{=} \{ g \mid g \in S_k \land m_i(f) \simeq m_i(g), \ i = 1 \ldots n, \ m_i(f) \in M_f \}
\]

This represents a necessary condition: the \( \simeq \) operator was used to state that \( g \) metric values, \( m_i(g) \), may be chosen to meet the specific goal. To identify similar functions, a threshold may be adopted:

\[
m_i(f) \simeq m_i(g) \Rightarrow (m_i(f) \leq m_i(g) \leq \theta_u(i) \land \\
\theta_l(i) \leq m_i(g) \leq m_i(f)), \ i = 1 \ldots n \ m_i(f) \in M_f
\]

where \( \theta_l(i) \) and \( \theta_u(i) \) are the \( i \)-th lower/upper bounds. Inside this range of values, \( g \) is considered to be a clone of \( f \). Clearly, to collect exact, or nearly exact, function duplicates, \( \simeq \) is implemented by the equality operator.

### 2.3.2.5 Measurement of the Cloning Ratio

Given two different software systems, say \( A \) and \( B \), information about the cloning extent between such software systems can be measured in terms of CR. The CR between \( A \) and \( B \) is defined as the ratio of the number of functions belonging to \( A \), having \( |C_f| \neq 0 \) when compared to functions in \( B \), to the number of functions contained in \( A \). In other words, it is the ratio of \( A \) functions having clones in \( B \) to \( A \) size. It should be noted that, according to the definition above, and due to the possibly different number of functions in \( A \) and \( B \), the CR of \( A \) to \( B \) may be different from the CR of \( B \) to \( A \).
2.4 Predicting Evolution via Time Series

This section details how TS can be applied to predict characteristics of future releases of a software system.

As said, the proposed approach is a black box approach, it that it models the evolution of a system in terms of directly observable characteristics, such as metrics for a software system. Conversely, other approaches model evolution by taking into consideration also process variables like programmer skills, cost, and social or technical constraints. Black box models rely on the assumption that software metrics capture essential characteristics of software artifact evolution. From this point of view, the ordered sequence of values of a software metric, can be considered as an instance of an univariate TS, which consists of sequentially recorded scalar observations. Given a sequence of observed values of a software metric (the observed TS), a black box model describing the TS dynamics may be identified and future values of metrics may be predicted.

Other prediction techniques, such as the state-space models, are available in literature (see [44] for further details). The drawback of the state-space approach is that, to identify these models, a state-space must be defined. Additionally, in conflict with the principle of a parsimonious use of the available observations, several matrices need to be estimated. Suppose that we would like to model a bivariate TS such that each observation is a two-dimensional array, where the first component is the software system size and the second component the software average cyclomatic complexity. A 4x4 matrix needs to be estimated from an observed TS, and the identification of the bivariate model parameters (the 4x4 matrix elements) requires more observations than the identifications of two independent univariate models. Another drawback of building complex models accounting for several explanatory variables is the difficulty of finding relationships between causes and effects. Consequently, simple univariate models, especially if a limited number of observations are available, should be adopted.

The proposed approach may be thought as a sequence of three steps, as follows:
1. Metrics extraction;

2. ARIMA model building; and

3. Predicting evolution and model validation.

2.4.1 Metrics Extraction

Different metric, namely LOCs, cyclomatic complexity, number of functions, as well as the CR have been extracted from each release of the software systems using Perl scripts. Software evolution gives raise to sequences of metrics, e.g., LOCs for several releases of a system, which can be thought of as a TS.

2.4.2 Model Building

A wide variety of approaches exist for predicting future values of a TS. In this work the TS was modeled with an ARIMA model, and the Box-Jenkins approach was applied to determine the model parameters. Statistical analysis of the data was performed with the freely available R environment [45] and, in particular, the time series package was adopted for TS modeling and prediction.

2.4.3 Predicting Evolution and Model Validation

To predict future values of a feature, a training TS was extracted from the observed TS related to the feature under study (e.g., size measure in LOC, number of functions). Afterwards, an ARIMA model was built on the training TS to predict metric future values. The predicted values were matched against the actual values and the method performance was assessed in terms of prediction errors. Given the observed TS $X_t$ with $t = 0, \ldots, N$ sub-series may be defined as:

$$tts_k = X_0, \ldots, X_k$$

(2.11)

The k-th steps ahead predicted value $\hat{x}_{T+k}$ is compared against the actual value (i.e., $X_{T+k}$) in order to evaluate the k-th steps ahead prediction error. Let $\hat{x}_T$ then be the predicted value at time T and $x_T$ its actual value, the MRE (Magnitude of Relative Error) is defined as:
\[ MRE_T = 100 \frac{\text{abs}(\hat{x}_T - x_T)}{x_T} \]  

(2.12)

The MMRE (Mean Magnitude Relative Error) over a sequence of \( n \) experiments is defined as:

\[ MMRE = \frac{1}{n} \sum_T MRE_T \]  

(2.13)

2.5 Case Studies

Analyses reported in this chapter have been performed on two open source software systems: the Linux Kernel and mSQL. Metrics prediction was performed on the Linux Kernel. Then, after detecting clones, a deeper analysis was performed to analyze clone evolution. While for the Linux Kernel the work was limited to analyze how new clones were introduced in new versions and old clones were factored out (thus keeping the CR almost stable), the TS approach was applied to mSQL for predicting clone evolution.

2.5.1 The Linux Kernel

Linux is a Unix-like operating system that was initially written as a hobby by a Finnish student, Linus Torvalds [46]. The first Linux version, 0.01, was released in 1991. Since then, the system has been developed by the cooperative effort of many people, collaborating over the Internet under the control of Torvalds. In 1994, version 1.0 of the Linux Kernel was released.

The peculiar characteristics of the Linux Kernel make it an ideal candidate as testbed for automated code examination and comprehension tools. It is based on the Open Source concept, and so there are no obstacles to discussing its implementation. It is not toy software, but one that is representative of real-world software systems. In addition, it is decidedly too large to be examined manually.

Unlike other Unices (e.g., FreeBSD), Linux it is not directly related to the Unix family tree, in that its kernel was written from scratch, not by porting existing Unix
source code. The very first version of Linux was targeted at the Intel 386 (i386) architecture. At the time the Linux project was started, the common belief of the research community was that high operating system portability could be achieved only by adopting a microkernel approach. The fact that now Linux, which relies on a traditional monolithic kernel, runs on a wide range of hardware platforms, including palmtops, Sparc, MIPS and Alpha workstations, not to mention IBM mainframes, clearly points out that portability can also be obtained by the use of clever code structure.

<table>
<thead>
<tr>
<th>Release Series</th>
<th>Initial Date</th>
<th>Number of Releases</th>
<th>Time to Start of Next Release Series</th>
<th>Duration of Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>9/17/91</td>
<td>2</td>
<td>2 months</td>
<td>2 months</td>
</tr>
<tr>
<td>0.1</td>
<td>12/3/91</td>
<td>85</td>
<td>27 months</td>
<td>27 months</td>
</tr>
<tr>
<td>1.0</td>
<td>3/13/94</td>
<td>9</td>
<td>1 month</td>
<td>12 months</td>
</tr>
<tr>
<td>1.1</td>
<td>4/6/94</td>
<td>96</td>
<td>11 months</td>
<td>11 months</td>
</tr>
<tr>
<td>1.2</td>
<td>3/7/95</td>
<td>13</td>
<td>6 months</td>
<td>14 months</td>
</tr>
<tr>
<td>1.3</td>
<td>6/12/95</td>
<td>115</td>
<td>12 months</td>
<td>12 months</td>
</tr>
<tr>
<td>2.0</td>
<td>6/9/96</td>
<td>34</td>
<td>24 months</td>
<td>32 months</td>
</tr>
<tr>
<td>2.1</td>
<td>9/30/96</td>
<td>141</td>
<td>29 months</td>
<td>29 months</td>
</tr>
<tr>
<td>2.2</td>
<td>1/26/99</td>
<td>19</td>
<td>9 months</td>
<td>still current</td>
</tr>
<tr>
<td>2.3</td>
<td>5/11/99</td>
<td>60</td>
<td>12 months</td>
<td>12 months</td>
</tr>
<tr>
<td>2.4</td>
<td>1/4/01</td>
<td>21</td>
<td>–</td>
<td>still current</td>
</tr>
<tr>
<td>2.5</td>
<td>22/11/01</td>
<td>71</td>
<td>–</td>
<td>still current</td>
</tr>
</tbody>
</table>

Table 2.1: Linux Kernel’s most important events.

Linux is based on the Open Source concept: it is developed under the GNU General Public License and its source code is freely available to everyone.

The most peculiar characteristic of Linux is that it is not an organizational project, in that it has been developed through the years thanks to the efforts of volunteers from all over the world, who contributed code, documentation and technical support. Linux has been produced through a software development effort consisting of more than 3000 developers distributed over 90 countries on five continents [47]. It should be noted that, due to the nature of the decentralized, voluntary basis development effort, no formalized development processes has been adopted, and thus it is worth investigating the quality characteristics of the resulting software.
An important management decision was establishing, in 1994, a parallel release structure for the Linux Kernel. Even-numbered releases were the development versions on which people could experiment with new features. Once an odd-numbered release series incorporated sufficient new features and became sufficiently stable through bug fixes and patches, it would be renamed and released as the next higher even-numbered release series and the process would begin again. The principal exception to this release policy has been the complete replacement of the O.S. virtual memory system in the 2.4 version series (i.e., within a stable release). The whole story, which has also led to the birth of alternative kernel trees, is dealt with in [48, 49]. At the time of writing (April, 2002), the latest kernel releases are 2.4.18 (stable) and 2.5.8 (experimental).

Linux Kernel version 1.0, released in March 1994, had about 175,000 lines-of-code. Linux version 2.0, released in June 1996, had about 780,000 lines-of-code. Version 2.4, released in January 2001, has more than two MLOCs. The current 2.4.18 release is composed of about 14000 files; its size is about 3 MLOCs (.c and .h). Counting the LOCs contained in .c files (i.e., excluding include files), its size is about 2.5 MLOCs (.c files only). The architecture-specific code accounts for 422 KLOCs. In platform-independent drivers (about 1800 files) there are about 1.6 MLOCs. The core kernel and file systems contain 12 KLOCs and 235 KLOCs respectively.

Table 2.5.1, which is an updated version of the one published in [47], shows the most important events in the Linux Kernel development time table, along with the number of releases produced for each development series.

Figure 2.5.1 reports Linux stable and experimental version evolution. Once a new stable version is released (e.g., 1.2.0, 2.0, 2.2.0 etc.), subsequent stable releases are developed from it. At a certain point an experimental version is released (1.3.0, 2.1.0, etc.), and subsequent experimental releases are developed from it, in parallel with the evolution of a stable version. That concurrent evolution continues until a new stable version is released. It is worth noting that the new version may encapsulate contribution for both latest stable and experimental releases, or only from one of them. Moreover, in some cases, some features incorporated into an experimental
release are successively given up. Finally, sometimes a stable version may evolve in parallel with another stable version (this is the case of 2.2 and 2.4).

![Diagram of Linux Kernel stable and experimental releases evolution]

Figure 2.3: Linux Kernel stable and experimental releases evolution.

2.5.2 mSQL

mSQL is a relational database system; it is developed in C and distributed by Hughes Technologies (http://www.hughes.com/au). mSQL relevant features evolution are summarized in Figure 2.5.2. In particular, the figure reports the evolution in terms of the number of procedures (i.e., C functions) and KLOCs of the 27 subsequent versions\(^1\) of mSQL. The first point on the x-axis is related to mSQL 1.0.6 while the last point is concerned with mSQL 2.0.10.

mSQL offers a subset of SQL (Structured Query Language) and its query interface. The first generation product (i.e., mSQL 1.0) was designed to provide high-speed access to small data sets using very few system resources on an average UNIX workstation.

\(^1\)The word version is used according to the IEEE Standard Glossary of Software Engineering Terminology (610.12-1990)
Several versions of mSQL have been available in various forms since June 1994, and the software system has undergone many enhancements becoming a very popular and a stable database system for small databases. mSQL 2 is a major redesign, as it can be easily deduced by Figure 2.5.2: the size and the number of functions were doubled; moreover, new builds (i.e., executables) were added.

![Graph showing number of procedures and KLOCs over versions](image.png)

**Figure 2.4: mSQL KLOCs and procedure number evolution.**

The new implementation goes beyond the initial design goals of mSQL 1.0 and provides functionality suited to larger applications. Moreover, the mSQL 2.0 server has been redesigned to execute multiple queries at the same time. Finally, executables in the distribution grew from 8 to 12: mSQL 2.0 has new functionalities to import/export data and is endowed with W3-mSQL 2.0, the second generation web interface package (i.e., w3auth, w3mysql, lite). The new W3-mSQL code provides a complete scripting language, with full access to the mSQL API (Application Programming Interface), within an HTML tag.
2.6 Predicting the Evolution: Results

To evaluate the accuracy of the TS modeling, product metrics (i.e., size in terms on KLOCs and number of functions, average cyclomatic complexity) were considered. In particular, two kind of TS were analyzed:

1. The stable releases of the Linux Kernel (88 releases from 1.0.0 up to 2.4.3);

2. The Linux Kernel overall evolution: once a stable version is released, subsequent stable releases may appear, until the next experimental version began. Then experimental releases are considered until the next stable version. This TS corresponds to following the black arrows drawn in Figure 2.5.1 and in the remainder will be referred as *timeline releases* (436 releases from 1.0.0 up to 2.4.3).

In all the cases, an initial training set of 20 points was used, and the ARIMA models were built according to the AIC criteria. The residual errors on ARIMA model parameters were kept under 10% of the parameter values.

The Linux Kernel evolution in terms of KLOCs and cyclomatic complexity for the *timeline releases* is shown in Figure 2.6 and 2.6. Figure 2.6 confirms the exponential growth of the software system size, in agreement to Lehman’s laws. The absence of significant steps highlighted how, considering also experimental releases when analyzing the evolution as explained above, gives an idea of the graceful evolution of the system. The number of function evolves in a way similar to KLOCs.

On the other hand, the average cyclomatic complexity is stable (it oscillates between 5.2 and 6.6). This, as also confirmed by the considerations in Section 2.7, indicates that programmers added new functionalities and/or maintained the system without increasing so much the complexity of the existing functions, but, instead, adding others and refactoring the code if necessary.

The analyses of the evolution on stable releases (# of functions evolution, similar to the KLOCs evolution, is plotted in Figure 2.6, while the average cyclomatic complexity is plotted in Figure 2.6). The first Figure clearly highlights the steps
Figure 2.5: Linux Kernel evolution in terms of KLOCs: timeline releases.

Figure 2.6: Linux Kernel evolution in terms of cyclomatic complexity: timeline releases.
between versions, indicating that at least some features “experimented” in the odd series were incorporated. The average cyclomatic complexity is, also in this case, almost constant, even if a slight improvement (i.e., refactoring) is remarkable between a version and the subsequent ones.

Figure 2.7: Linux Kernel evolution in terms of # of functions: stable releases.

MREs were computed, on each TS, for different steps ahead predictions (ranging from one to ten). Some results for the timeline releases are shown in Figure 2.6 and 2.6 for one, five and ten steps ahead predictions. Generally contained within 10%, MREs exhibit peaks in correspondence (or in proximity) of a release change. For example, the maximum MRE ranges from 10% of one-step-ahead prediction to 51% of ten steps ahead KLOCs prediction. Although a TS of timeline releases tried to follow the complete evolution of the Linux Kernel across both stable and experimental releases, prediction errors in correspondence of version changes highlighted the fact that a new version was released when a relevant number of features was incorporated.

Higher errors were obtained when performing predictions on the sequence of stable releases. Figures 2.6 and 2.6 reports MREs for number of functions and cyclomatic
Figure 2.8: Linux Kernel evolution in terms of cyclomatic complexity: stable releases.

Figure 2.9: Timeline releases - MRE on KLOCs prediction.
complexity prediction. As expected, these errors were obtained in correspondence of a new version. Even for the one step ahead prediction, the MRE was 57% in correspondence of version 2.0, and 46% in correspondence of version 2.2. However, the corresponding increase in terms of number of functions was of 136% and 85% respectively, therefore the error can be considered acceptable.

The highest evolution steps (and thus prediction errors) may be traced back mainly to the following changes introduced in Kernel 2.0:

- Expanded platform support;

- Expanded IDE, SCSI devices and Ethernet adapters support;

- Module support;

- Several network-related improvements (SMB, IP tunneling, IP masquerading, IP multicast routing, ISDN support);

- Support for new filesystems (HPFS, Amiga, UMSDOS, VFAT and others); and
Figure 2.11: Stable releases - MRE on \# of functions prediction.

Figure 2.12: Stable releases - MRE on cyclomatic complexity prediction.
• Kernel thread support.

and in Kernel 2.2.0:

• Several improvements were added to networking (firewalling, routing, traffic bandwidth management);

• The TCP stack was improved;

• New filesystems were added;

• The NFS daemon was improved; and

• The sound configuration was improved, and support for new soundcards added.

Figures 2.6 and 2.6 summarizes the MMREs obtained, for different steps ahead predictions, predicting metrics on timeline releases and stable releases respectively. For timeline releases prediction, even the 10 steps ahead MMRE is very low (about 3.5% and 4% for LOCs and # of functions respectively). The errors significantly increase for prediction on stable releases TS even if, as said, can be considered as acceptable provided the amount of change introduced with a new stable version.

2.7 Observing the Cloning Evolution of the Linux Kernel

This section aims to study the evolution of the amount of cloned code in a large, multi-platform, multi-release software system. Nineteen releases of a multi-million lines-of-code software, the Linux Kernel (releases 2.4.0 through 2.4.18), have been used as case study.

The evaluation of the cloning extent has been performed at different levels. Clones have been identified among top-level directories of the source tree, which essentially correspond to major subsystems. Furthermore, the same analysis has been performed between non top-level directories at the same nesting level of the source tree, i.e., within major subsystems.

In particular, the experimental activity carried out has addressed the following research questions:
Figure 2.13: Timeline releases - metrics prediction MMREs.

Figure 2.14: Stable releases - metrics prediction MMREs.
• Which is the cloning extent within the Linux 2.4.x kernel major subsystems?

• Which is the cloning extent within the subsystems related to the different supported platforms?

• Is there a trend in CR when the system evolves?

The results reported in the following were computed using a slightly different procedure as compared to the one followed in [27]. In [27], the clones were identified considering all functions contained in the system regardless of function sizes (measured as the number of LOCs of the function body). Doing so, small functions (e.g., functions setting or getting the value of a structure) very often cluster together. However, it may be argued that these functions do not really represent clones, and thus that the resulting CR is biased by false positives.

To study the influence of short functions on CR, this index was computed for two different configurations. The first configuration corresponds to the assumptions made in [27]; namely, all functions, regardless of their size, were considered. In the second configuration, instead, all functions with a body shorter than five LOCs were discarded, detecting clone clusters and computing the CR only on the remainder.

Analyzing CRs on several Linux releases, it was noticed that CRs among all possible combinations of Linux subsystems were often null or very low, thus leading to sparse cloning matrices. Furthermore, according to the definition of CR, high CR value does not necessarily imply high number of replicated code snippets. A 50% CR may correspond just to a couple of cloned functions, if small subsystems are considered. On the other hand, if the analyzed subsystems contain a high number of functions, say 1000, even a CR as low as 1% is worth to be considered. The following subsections report results that were considered significant either in relative (e.g., high CR values) or in absolute terms (e.g., high number of cloned functions).

After reporting a detailed analysis on clone of release 2.4.18, investigations on the cloning evolution starting from release 2.4.0 will be reported.
2.7.1 Kernel 2.4.18 Analysis

The experimental activity presented in this subsection was driven by the first two research questions above specified, i.e., computing the CR among major Linux architectural components and the percentage of duplicate code among different supported platforms. However, in the authors’ knowledge, it does not exist any documentation of the Linux architecture, in that no document describes the system at a high level of abstraction. Bowman et al. derived both the conceptual architecture (the developers’ system view) and the concrete architecture (the implemented system structure) of the Linux Kernel [50, 51, 52]. They started from a manual hierarchical decomposition of the system structure, consisting of the assignment of source files to subsystems, and of subsystems hierarchically to subsystems. As shown in [52], most of the times the extracted subsystems correspond to directories in the source code tree. For simplicity’s sake, in the analysis performed, it has been assumed that each directory of the source tree contains a subsystem (at a proper level of the system hierarchy). Thus, the search for cloned code was performed by comparing the code contained in any two directories.

Figure 2.15 shows two different examples of function clones identified. The first clone pair (top of the figure), is an example of a function copied from mips to mips64 memory management subsystem. The second clone pair is instead a cross-system example: although the accessed data structure has different field names, the action actually performed is the same, i.e., the removal of an item from a concatenated list.

Table 2.7.1 reports the CRs higher than 1% among Linux major subsystems (i.e., the twelve top-level directories, documentation and include directories excluded). CRs are reported along with the corresponding number of cloned functions, for both the considered configurations (i.e., functions longer than five LOCs, and all functions).

Observing Table 2.7.1, it can be easily recognized that:

- The table contains only seven rows, out of 144 possibilities; in other words, only very few subsystem comparisons gave raise to appreciable clone extents;
Figure 2.15: Two examples of clones found.

- The difference between the results obtained considering all functions and those obtained with a 5-LOCs threshold is relevant; and

- Though CRs among major subsystems is not very high, even a small ratio (e.g., 1.43% between arch and drivers) corresponds to a non-negligible (152) number of cloned functions, as these subsystems are very large.

It is worth pointing out that in the two configurations CRs were computed considering the ratio to the total number of retained functions. This may lead to two counterintuitive phenomena: higher CR for functions ≥ 5 LOCs, and different CRs corresponding to the same number of cloned functions, because of the lower number of functions that are assumed to belong to the system.

A similar approach was followed to evaluate the cloning extents within the sub-systems related to the different supported platforms. The arch directory contains fifteen sub-directories, each one corresponding to a supported processor architecture
<table>
<thead>
<tr>
<th>Subsystems Compared</th>
<th>Functions ≥ 5 LOCs</th>
<th>All Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CR</td>
<td>Functions Cloned</td>
</tr>
<tr>
<td>arch-drivers</td>
<td>1.43%</td>
<td>152</td>
</tr>
<tr>
<td>fs-drivers</td>
<td>2.06%</td>
<td>93</td>
</tr>
<tr>
<td>ipc-arch</td>
<td>1.45%</td>
<td>1</td>
</tr>
<tr>
<td>kernel-arch</td>
<td>2.11%</td>
<td>114</td>
</tr>
<tr>
<td>lib-arch</td>
<td>2.90%</td>
<td>9</td>
</tr>
<tr>
<td>lib-net</td>
<td>1.45%</td>
<td>4</td>
</tr>
<tr>
<td>mm-drivers</td>
<td>1.36%</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2.2: CRs ≥ 1% among major subsystems.

(e.g., i386, s390, sparc). Each platform has, among others, its own kernel and memory management mm implementations. In particular, Table 2.7.1 shows the CRs among mm for the architectures supported by Linux 2.4.18. A different threshold (10%), higher than the 1% used for Table 2.7.1, was used to avoid reporting meaningless data. Only 10 rows out of 225 were retained and, as it can be readily seen in Table 2.7.1, the mm subsystems contain only few cloned functions even if the CR values are non-negligible.

CRs were also computed for the core kernel subsystem (e.g., arch/i386/kernel versus arch/ppc/kernel). Those results were not presented here since, in a very similar way to mm, even if some architectures exhibit relevant CRs the number of cloned functions were very low (often one, sometimes two or three).

The data for Linux 2.4.18 confirmed the results obtained on different Linux releases [27, 52]. In most cases, the implementation of similar functionalities was carried out by resorting to code reuse (function dependencies across different subsystems) rather than to cloning. This is clearly shown by the small number of subsystem comparisons exhibiting a non-negligible number of cloned functions.

There are few exceptions, however. Among these, the CR between the mips64 and mips mm subsystems (22.61%, with six cloned functions). The ratio obtained without filtering out functions smaller than five LOCs was slightly higher (28.57%), but considerably smaller than the 38.4% computed on the Linux Kernel 2.4.0 and reported in [27]. However, even in this case, the absolute number of cloned function
Table 2.3: CRs ≥ 10% among mm architecture dependent code. 

is low.

Table 2.7.1 reports data on CR and cloned functions among Linux drivers. Driver subsystems (e.g., the SCSI and IDE drivers, the char and USB or the PCI drivers) are the largest part of the kernel code and are subject to continuous evolution. CR among driver subsystems is fairly low, and in general only few functions are duplicated. An exception seems to be the number of duplicated functions between the char and sbus subsystems, where 53 clone clusters were identified.

Table 2.4: CRs ≥ 5% among drivers.
2.7.2 Analyzing Cloning Evolution

This section aims to investigate how the cloning ratio varied in the Linux Kernel from release 2.4.0 to release 2.4.18. The analysis has been performed at different levels of granularity:

1. The overall cloning on the entire Linux Kernel;

2. The cloning among major subsystems; and

3. The cloning among architecture-dependent code of some subsystems.

![Graph of Common Ratio (%)](image)

**Figure 2.16: Overall evolution of CR.**

Figure 2.16 reports the evolution of the overall CR, computed considering both all functions and only functions ≥ 5 LOCs. The figure shows that results are quite different if small functions are filtered. In both cases, the cloning variation over releases is not relevant. Focusing our analysis on CR for functions ≥ 5 LOCs (as well as in all the further analyses presented in this subsection), the CR varies from 14.33% to 16.11% (i.e., a maximum difference of about 2%) and its standard deviation
is 0.03. This supports the hypothesis that no considerable refactoring was performed across 2.4.x releases.

The analysis of CR evolution among major subsystems confirms the previous impressions. Even in this case, no relevant change in the CR has been detected (variations are less than 2%). Figure 2.17 shows the evolution of cloning between \texttt{fs} and \texttt{mm} subsystems. It is worth noting that, from release 2.4.0 to release 2.4.4, the CR in \texttt{mm} decreased of about 1.6\% (about 20 functions), indicating a possible refactoring activity.

![Figure 2.17: Evolution of CR between mm and fs.](image)

In a way similar to the results presented in the previous subsection, the most interesting behavior of CR evolution was found inside the \texttt{mm} subsystem, and in particular between the \texttt{mips64} and \texttt{mips} architecture-dependent code. The values of CR are plotted in Figure 2.18. The figure shows that the CR ranged from 37.68\% for release 2.4.0 (slightly different from the 38.4\% reported in [27], and computed considering all functions) to 22.60\% for release 2.4.18.

One may argue that the programmers first ported the \texttt{mm} subsystem to the \texttt{mips64} architecture by cloning portions of the \texttt{mips} code, and then performed a refactoring.
Figure 2.18: Evolution of CR between mips and mips64 code inside the mm subsystem.

However, a more detailed analysis demonstrated the exact contrary. In fact, the number of functions (≥ 5 LOCs) composing the mips64 portion of mm varied from 69 in release 2.4.0 to 115 in release 2.4.18, in that the number of cloned functions remained constant to:

$$37.68\% \text{ of } 69 = 22.60\% \text{ of } 115 = 26$$

In other words CR, as much like any relative measure, should be used with great care, always resorting to the examination of absolute values.

### 2.7.3 Answers to Questions on Cloning Evolution

The answers to the research questions can therefore be summarized as follows:

- CR among sub-systems can be considered at a physiological level;

- Recently introduced architectures tend to exhibit a slightly higher CR. The reason for this is that a subsystem for a new architecture is often developed incrementally with respect to a similar one (e.g., mips64 from mips); and
• The evolution of CR, at the overall level, tends to be fairly stable, thus suggesting that the software structure is not deteriorating due to copy-and-paste practice.

It is worth to point out that almost always even a relatively high CR value does not represent a remarkable number of duplicated functions. Code duplication may be considered relevant only among few major subsystems (e.g., arch versus drivers). But even in this case, due to the high number of functions in the subsystems, a CR value of about 1-2% ends up in just 100-150 duplicated functions.

2.8 Predicting Cloning Evolution

In this section an approach for monitoring and predicting cloning evolution, more precisely the average number of clones per function across subsequent versions of the same software system, is presented. Monitoring and predicting cloning evolution can be used at least in two different applicative scenarios within the software maintenance field. In particular:

1. The actual effort to evolve and maintain a system is likely to vary depending on the amount of clones found in a system. Thus, the ability to predict clones evolution can be used to define the effort required by the future maintenance activities; and

2. A discrepancy between the actual values and the predicted ones is a parameter that may give an a-posteriori validation of a certain evolution hypothesis.

Cloning evolution across subsequent versions of the same software system can be modeled using TS. Given a set of subsequent versions of a software system, the average number of clones per function in each version can be thought of as a TS; thus a predictive model may be identified.

The proposed approach has been applied to 27 subsequent versions of mSQL. The time span period of the analyzed mSQL versions covers four years, from May 1995 (mSQL 1.0.6) to May 1999 (mSQL 2.0.10).
### 2.8.1 Case Study Results

<table>
<thead>
<tr>
<th>mSQL 1.0.x</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ident.</td>
<td>$x_a$</td>
<td>$x_b$</td>
<td>$x_c$</td>
<td>$x_d$</td>
<td>$x_e$</td>
<td>$x_f$</td>
<td>$x_g$</td>
<td>$x_h$</td>
<td>$x_i$</td>
<td>$x_j$</td>
</tr>
<tr>
<td>Actual</td>
<td>1.129</td>
<td>1.127</td>
<td>1.135</td>
<td>1.122</td>
<td>1.130</td>
<td>1.122</td>
<td>1.123</td>
<td>1.121</td>
<td>1.130</td>
<td>1.168</td>
</tr>
<tr>
<td>MRE (%)</td>
<td>-</td>
<td>-</td>
<td>0.60</td>
<td>0.67</td>
<td>0.31</td>
<td>0.05</td>
<td>0.07</td>
<td>0.03</td>
<td>0.05</td>
<td>3.75</td>
</tr>
</tbody>
</table>

Table 2.5: Actual TS Values MRE (mSQL 1.0.x).

<table>
<thead>
<tr>
<th>mSQL 2.0,β</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ident.</td>
<td>$x_{a1}$</td>
<td>$x_{a2}$</td>
<td>$x_{a3}$</td>
<td>$x_{a4}$</td>
<td>$x_{a5}$</td>
<td>$x_{a6}$</td>
<td>$x_{a7}$</td>
</tr>
<tr>
<td>Actual</td>
<td>1.44</td>
<td>1.43</td>
<td>1.43</td>
<td>1.38</td>
<td>1.36</td>
<td>1.33</td>
<td>1.41</td>
</tr>
<tr>
<td>MRE (%)</td>
<td>18.82</td>
<td>19.59</td>
<td>0.03</td>
<td>3.95</td>
<td>1.43</td>
<td>1.79</td>
<td>5.88</td>
</tr>
</tbody>
</table>

Table 2.6: Actual TS Values and MRE (mSQL β releases).

<table>
<thead>
<tr>
<th>mSQL 2.0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ident.</td>
<td>$x_{a17}$</td>
<td>$x_{a18}$</td>
<td>$x_{a19}$</td>
<td>$x_{a20}$</td>
<td>$x_{a21}$</td>
<td>$x_{a22}$</td>
<td>$x_{a23}$</td>
<td>$x_{a24}$</td>
<td>$x_{a25}$</td>
<td>$x_{a26}$</td>
</tr>
<tr>
<td>Actual</td>
<td>1.45</td>
<td>1.40</td>
<td>1.43</td>
<td>1.28</td>
<td>1.45</td>
<td>1.40</td>
<td>1.41</td>
<td>1.44</td>
<td>1.43</td>
<td>1.44</td>
</tr>
<tr>
<td>MRE (%)</td>
<td>2.19</td>
<td>3.87</td>
<td>2.64</td>
<td>11.96</td>
<td>11.82</td>
<td>0.06</td>
<td>2.14</td>
<td>3.04</td>
<td>0.47</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Table 2.7: Actual TS Values and MRE (mSQL 2.0.x).

The average number of clones per function was evaluated for each mSQL analyzed version according to the clones detection technique previously described in Section 2.3.2. Such data, shown in the row labeled Actual of Table 2.8.1, Table 2.8.1 and Table 2.8.1, can be thought of as a collection of observations made sequentially in time, and thus modeled using TS.

The experimental activity followed a cross validation procedure [53]; in particular 25 experiments ($Exp_1, Exp_2, \ldots, Exp_{25}$) were run: in each experiment a training TS was extracted from the observed TS related to the average number of clones per function; then, the training TS was analyzed and modeled to predict its future values; finally the predicted values were matched against the actual values and the method performance was assessed in terms of one step ahead MRE (shown both in Table 2.8.1, Table 2.8.1, Table 2.8.1 and plotted in Figure 2.19).

As previously stated, monitoring and predicting clones evolution can be used at least in two different applicative scenarios and the available data support such a
conjecture. In particular, the actual effort to evolve and maintain a system is likely to vary depending on the amount of clones found in a system. For this perspective, predicting the average number of clones per function may give a significant support in defining the actual effort required to maintain and evolve such a new software system version. It is worth noting that the one step ahead MMRE was 3.81%: on the entire sequence of the mSQL analyzed versions, clones evolution was predicted with a quite acceptable error.

The analysis of the change logs related to the analyzed mSQL versions highlighted that during the evolution from mSQL 1.0.6 to mSQL 1.0.16, the principal activity was concerned with bug fixing, without significant changes occurred at design and code level. On the other hand, dramatic changes were introduced in mSQL 2.0.B1: old features were redesigned and new functionalities were added. The following β versions (i.e., from mSQL 2.0.B2 up to 2.0.B7.1) were devoted to fixing the new
discovered bugs. Also the versions from 2.0.1 to 2.0.3 were mainly concerned with bug fixing while significant changes where introduced with the 2.0.4. In particular, in mSQL 2.0.4 a partial remodularization was performed to improve search functionalities. Finally, bugs fixing was the principal activity in the remaining analyzed versions (i.e, from mSQL 2.0.5 up to 2.0.10.1) too.

The analysis of the one step ahead MRE curve, shown in Figure 2.19, highlights two peaks. The first peak appears with the predicted values related to the first two \( \beta \) versions (2.0.B1 and 2.0.B2); the other peak appears with the prediction related to mSQL 2.0.4. According to available data, whenever major changes occurred the predicted values were affected by an error bigger than 11\%, while in all the other cases the error is less than 6\%. Thus, a discrepancy between the actual and the predicted values is a parameter that may give an \textit{a-posteriori} validation of a certain evolution hypothesis.

For example, whenever a customer should acquire a new version of a software system he/she could be interested in the evaluation of the evolution extent; such evaluation can be supported by matching predicted values with the actual values of the average number of clones per function. In other words, when a new version supposed to implement dramatic changes is delivered, the discrepancy between the predicted and actual values may be an indicator of the real extent of the changes.

2.9 Conclusions

A TS-based approach has been applied to predict the evolution of the Linux Kernel. The average prediction errors were generally low and, though with some relevant peaks, they can be considered very good. Moreover, the considerably lower MMREs reported for \textit{timeline releases} analysis revealed that, to account for the real evolution of the Linux Kernel, experimental releases should be considered; the absence of peaks in timeline MREs is also supporting the hypothesis that most of the innovations introduced into experimental releases are incorporated into the next stable version.

The proposed approach can effectively help the staffing of a software development
project, even if the choice of the prediction method (i.e., ARIMA models) should be
done carefully, since not all categories of software systems exhibit behaviors suitable
to prediction by means of ARIMA models.

The CR for several releases of the Linux Kernel has been measured, discussing
the process and the strategies that can be adopted to analyze a large multi-platform,
multi-million lines-of-code real word software system. Software metrics at function
level were extracted and duplicate code among kernel subsystems detected. Different
thresholds were adopted to extract the CR, to avoid biased results due to false
positives induced by small functions. In the present study, two configurations were
considered, the first corresponding to the analysis of all functions belonging to the
system, the second discarding the functions with a body shorter than five LOCs.

Linux has not been developed through a well-defined software engineering process,
but by the cooperative work of relatively uncoordinated programmers. Nevertheless,
the overall CR, as well the CR of its subsystems, is remarkably low, especially if
small functions are not taken into account.

An approach for monitoring and predicting the evolution of clones across subse-
quent versions of a software system was finally presented. An experimental activity
was performed on 27 subsequent versions of mSQL, to empirically quantify the per-
formance of the approach itself. Results are encouraging: on average an error of
3.81% has been observed. The monitoring and prediction of cloning evolution may
be employed at least in two different applicative scenarios: the prediction may be
used to define the actual effort required to maintain and evolve future software system
versions. Moreover, the analysis of the mSQL change logs highlighted that a discrep-
ancy between the estimated and the actual values was due to dramatic changes in
the system. Thus, the discrepancy between the actual and the predicted values is a
parameter that may give an a-posteriori validation of a certain evolution hypothesis.

In conclusion, this chapter makes two main contributions: it proposes TS mod-
eling as an applicable approach within the software maintenance field and gives an
insight of clones evolution across subsequent versions of the same software system. In
fact, even if there have been many publications concerned with the techniques aimed
at identifying cloned components in a software system, few papers have studied clone
dynamics across several versions of the same software system. Future work will be
devoted to assess the performance of the prediction approach on different kinds of
software systems.
Bibliography


http://www.r-project.org.


http://plg.uwaterloo.ca/~itbowman/CS746G/a1/.


CHAPTER III

Analysis of Multi-Configuration Code

3.1 Introduction

Software systems are often developed with configurable capabilities, and to run under several different hardware and software platforms. Preprocessor conditionals are often used to configure the software accordingly. Setting the configuration variables and compiling gives a single configured instance of the program, and under ideal circumstances, each configurer can do just that and obtain a working program for his configuration.

However, full configurability is difficult to achieve in practice, and it can be impractical for the programmer to verify that every configuration works correctly. Knowing that every possible configuration is type-correct is a necessary first step to ensure that every configuration actually works. Testing a particular configuration for type-correctness is easily done if one has a compiler, by simply configuring and compiling. This approach is not effective once the number of configurations becomes significant: there should be a tool to verify that a program is type-correct in all possible configurations. This chapter defines an approach to capture what types are declared under what configurations, and to explore how these conditional types can be used to detect configuration-instance type faults [1].

The chapter is organized as follows. Section 3.2 and Section 3.3 explain the motivations for this work and summarize related work, respectively. Section 3.4 describes the architecture of the proposed tool. Data on the impact of preprocessor condition-
als on different software systems are reported in Section 3.5, before conclusions.

3.2 Preprocessor-Conditioned Types

```c
#include <stdio.h>

#define ALPHA 1
#define ATARI 2

if defined(ALPHA) || defined(ATARI)
    /* Use char here */
else if defined(pyr)
    char * buffer;
#else if defined(i386)
    int buffer;
#else;
    ...

    *buffer = 2;
```

Figure 3.1: Example of a preprocessor-conditioned declaration.

Figure 3.1 shows an example of preprocessor-conditioned variable declaration, and its use inside an expression. Depending on the configuration, the type of the variable `buffer` may be:

1. char: if the expression `(defined (ALPHA) || defined (ATARI)) && defined (pyr)` is true;

2. int: if the expression `!(defined (ALPHA) || defined (ATARI)) && defined (i386)` is true;

3. Undefined: in all other cases.

This may lead to two categories of problems:

- Inconsistent use: e.g., declared as `int` but used as pointer;

- Used but not declared: The system could be compiled in a configuration where the variable `buffer` is never defined but is actually used at a certain point, since the use has accidentally not been excluded by preprocessor condition similar to the one excluding the declaration.
For particular configurations, these problems are detected at compilation time. However, this solution is infeasible for systems in which the number of possible configurations is huge, and it is simply impractical to compile every configuration before release to customers. For example, the Linux 2.4.0 Kernel contains more than 7000 files, and runs on 10 different processors (see Section 2.3.2.1). It has 400 preprocessor switches, each of which may assume three different values (Y to include the code into the compiled kernel; N, or commented switch, to exclude the code, or M to produce a dynamically loadable module) drive the actual kernel configuration. This enables some $10 \times 400^3$ possible different configurations (not all possible combinations of switches make sense, e.g., it is unlikely for a machine having several sound boards installed), which should be considered when testing the system.

Furthermore, there may be some problems that are discovered only during execution, when the affected portion of code is reached. Most conditionals are inserted to compile the software system under different possible configuration. However, some are not and can effectively be ignored for type analysis. A typical (and, often, the most relevant) example is the preprocessor code used to avoid circular inclusion:

```
#ifndef MYFILE_H
#define MYFILE_H
#endif
```

Heuristics (often pattern-matching oriented) are adopted to avoid considering these preprocessor directives as conditions on variables.

### 3.2.1 Examples of Faults

In order to better understand what kind of faults we propose to discover, this section reports some examples of failed type-checking. Consider, for example, the following C code:

```c
{ #if ARRAY
    char **x;
#else
```
char *x;
#endif
x="Hello world!";
}

It is clear that, in the configuration ARRAY, x is a pointer-to-pointer to char, therefore the assignment will result as faulty.

Another example is the following:

{ #if M
    byte *address;
#endif
#if N
    char *address;
#endif
    printf("the address is:%s", address);
}

In this case the printf statement works correctly only in the N configuration.

3.3 Related Work

Programmers tend to consider it necessary to port C software from one configuration to another, as C runs on "practically everything". When differences among systems cause difficulties, the usual first solution adopted by programmers is to write two different versions of the code, one per system, and use #ifdef to choose the appropriate one. However, the large use of #ifdef to attempt at portability is usually a source of several problems. The result is usually an unreadable, and difficult to maintain software system [2].

Ernst et al. [3] presented the first empirical study of the use of the C macro preprocessor. They analyzed 26 packages, to determine the practical use of preprocessor directives. The authors also proposed a taxonomy of various aspects of preprocessor use. This paper reported data regarding the prevalence of preprocessor directives,
macro body categorizations, use of the C preprocessor to achieve features impossible in the underlying language, inconsistencies and errors in macro definitions and uses, and dependences of code upon macros. Hu et al. [4] coped with conditional compilation problems. They presented an approach based on symbolic execution of preprocessing directives. Their goal was to find the simplest sufficient condition to reach/compile a line of code containing a preprocessor directive, and the full condition to reach/compile it. Experiments have been conducted on Linux Kernel, using a tool that automates the approach presented.

Livadas and Small [5] addressed problems concerning source code containing preprocessor constructs, such as included files, conditional compilation, and macros. The authors proposed a mapping from token in the preprocessor output to the source file(s), and discussed the use of these correspondences, through an internal program representation, for maintenance purpose jointly with techniques including program slicing, ripple analysis, and dicing. Baxter and Mehlich [6] explained disadvantages in having preprocessor conditionals inside code when the presence of configuration dependent code loses its utility, and proposed a method for its removal. A method to simplify preprocessor Boolean expressions was adopted to easily evaluate expressions. Type inferencing is also present in other code-analysis tools. For example, TXL [7, 8] developers implemented a type inferencing and checking prototype as an attribute grammar.

3.4 Architecture of Type-Checking Tool

The tool was conceived to be implemented using the DMS (Design Maintenance Systems) [9], a reengineering vision and toolkit that enables the analysis, translation, and/or reverse engineering of software systems. DMS is composed of:

- Lexer;
- Parser;
- Attribute evaluator;
Figure 3.2: Architecture of the tool.

- PrettyPrinter;
- Rule Applier; and
- API to manage AST and symbol table.

The architecture of the proposed tool is shown in Figure 3.2. Once the source code of the system to analyze has been parsed and the AST produced, the *Conditional Name Resolver* builds an *Enhanced Symbol Table*, in which the type of each symbol is conditioned by an expression (see details in Section 3.4.1). Type-checking on the expressions contained in the AST of the software system is then performed for all the possible configurations.

One of the main advantages of the proposed tool is that, while with a traditional compiler can only compile and check the software system in only one possible configuration at a time, this tool can type-check each expression for all possible combination of types variables involved in the expression. Finally a report log, containing the list of all possible inconsistencies, is produced.
3.4.1 Building the Enhanced Symbol Table

DMS builds the symbol table of a source code file using an attribute evaluator named \textit{Name Resolver}. The DMS symbol table supports the definition of hierarchical maps from identifiers to values together with visibility restrictions for those identifiers. An example of symbol table is depicted in Figure 3.3.

![Symbol Table Diagram](image)

Figure 3.3: Example of DMS symbol table.

However, the standard DMS \textit{Name Resolver} for C does not take care of the preprocessor directives, unconditionally inserting symbols into the symbol table, or leaving to a preprocessor the task of expanding directives.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Scope</th>
<th>Type</th>
<th>Conditional Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>0</td>
<td>char*</td>
<td>defined (ALPHA)</td>
</tr>
<tr>
<td>buffer</td>
<td>0</td>
<td>int</td>
<td>!(defined(ALPHA)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>float</td>
<td>true</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>int</td>
<td>defined(SPARC)</td>
</tr>
</tbody>
</table>

Table 3.1: An example of enhanced symbol table.

What the \textit{Enhanced Symbol Table} aims to do is to associate, to each triple symbol-scope-type, a Boolean expression such as those in the examples of Section 3.2. Moreover, it is worth noting that, in cases such as Figure 3.1, the symbol table can contain several instances of the same couple symbol-scope, where the type varies
with the conditioning expression (see Table 3.1). For a declaration outside the preprocessor conditionals, such as variable x in the table, the Conditional Expression field is, obviously, set to true.

Starting from the original DMS Name Resolver, a new attribute expr was added to the attribute evaluator. This attribute brings the Boolean condition from the conditional directive to the variable declaration. More Boolean expressions are composed in presence of nested preprocessor conditionals.

At the root node of the AST the attribute expr is set to true (i.e., if a symbol is declared outside a preprocessor conditional, it always assumes the associated type, as in the original DMS symbol table). Figure 3.4 shows how the new attribute flows through the grammar terms.

A preprocessor-conditioned code block can be described by one of the following grammar rules:

1. block = if_group block ’#endif’
2. block = if_group block
   #else’ block ’#endif’
3. block = if_group block
   elif_group block
   ’#endif’
4. block = if_group block
   elif_group block
   ’#else’ block
   ’#endif’

where an if_group may be a directive:

5. if_group = #if expr
6. if_group = #ifdef IDENTIFIER
7. if_group = #ifndef IDENTIFIER

Let us consider now, as highlighted in Appendix A, the second case (for the other rules the behavior is similar). The if_group passes the symbolic Boolean expression
(expr) to the conditioned blocks, i.e., to both block grammar symbols in the right-side of the grammar rule (the expression is negated when passed to the second block, being the else branch of the condition).

The symbolic expression coming from the if_group is then combined with the expression coming form the outer block (i.e., from the left side of the grammar rule) by an AND operator, building a conditional expression for declarations inside nested preprocessor conditionals. Finally, the conditional expression is then simplified algebraically using rewrites. Further details about how expressions are composed and simplified are shown in Section 3.4.2.

Once the attribute expr reaches a symbol declaration, the insertion of the symbol
Figure 3.5: Transformation rules for composing expressions.

in the symbol table follows the schema explained by the pseudo-code shown in Appendix A: the function AddCondSymbol inserts all symbol information (name, scope, type), along with the conditional expression in the symbol table. If the declaration is not inside a preprocessor conditional, then the conditional expression assumes the Boolean value true (e.g., variable x in Table 3.1).

It is worth noting that, if the same symbol is already present in the same scope, then the function AddCondSymbol simply adds another instance of the couple type-conditional expression (e.g., the buffer variable in Table 3.1).

3.4.2 Composing and Simplifying Expressions

The DMS Rule Specification Language enables the specification of patterns and rules to compose and transform ASTs. For composing expressions, two transformation rules were used:

1. To negate an expression, before passing it to the else branch of a conditional, and

2. To compose nested conditions using the Boolean AND operator.

The DMS transformation rules are shown in Figure 3.5. The first rule defines a pattern, named not_expression. The expression in parentheses indicates that the AST node to be replaced must be of type simp_constant_expression; the type of the instantiated AST node is indicated after the colon. The expression in the right side indicates that the matched pattern (e1) will be transformed to !(e1). A new AST node will be instantiated, whose type (simp_constant_expression) is indicated in the left side of the expression after the colon. Similarly, the second rule composes
two AST nodes creating a `simp_logical_and_expression` node.

```java
private rule eliminate_parentheses_27 (e:simp_expression):  
simp_expression->simp_expression  
  ="(\(e\))"->"e\":simp_expression".

private rule simplify_not_not(e:simp_unary_expression):  
simp_unary_expression->simp_unary_expression  
  ="!!\(e\)"-> e.

private rule simplify_or_repetition_2(e1:simp_logical_and_expression,e2:simp_logical_and_expression):  
simp_logical_or_expression->simp_logical_or_expression  
  ="\(e1||\(e2||\(e2\)"->"e1||\(e2\".

private rule simplify_or_and_1a(e1:simp_logical_and_expression,  
e2:simp_logical_and_expression):  
simp_logical_or_expression->simp_unary_expression  
  ="\(e1||\(e1&\&\(e2\):simp_inclusive_or_expression"->"\(e1\):simp_unary_expression".
```

Figure 3.6: Examples of rules.

A further task performed by transformation rules is to simplify conditional expressions. Suppose we have a piece of code written as follows:

```c
#if defined(i386)
#if defined(i386) || defined (intel)
  int x;
#endif
#endif
```

The condition on the variable x should ideally be:

```c
#if defined(i386)
```

Simplifications are performed by a set of transformation rules, applying well-known properties of the Boolean algebra, and pruning superfluous parentheses. Some examples of rules are reported in Figure 3.6. Further details can be found in [6]. It is worth noting that these rules work properly given that associative and commutative properties have been properly defined in the grammar for all Boolean operators.

### 3.4.3 Type-Checking

In this subsection we will consider how the Enhanced Symbol Table can be extended with a type-checking mechanism. A mechanism for type-checking is similar
to the mechanism for evaluation. In evaluation, an expression is processed to arrive at a value, while an expression is processed to determine a type for type-checking. Indeed, a type-checker verifies that the type of a construct matches that expected by its context. For example, a type-checker verifies that a de-referencing operator is applied only to a pointer, or that indexing is done only on an array. Type-checking is typically achieved by providing a mechanism to define type-constraints indicating the type of a variable [10].

The design and the implementation of a type-checker for a specific language is based on information about the syntactic constructs in the language, the notion of the types, and the rules for assigning types to language constructs. A collection of rules, for assigning type expressions to the various part of a program, is required. The type of a language construct is referred in literature as "type expression". A type expression can be either a basic type or formed by applying a type constructor. The set of basic types and constructors depends on the specific language. For the C language all this information can be extracted from the C reference manual [11]. A convenient way to represent and evaluate a type expression is to use a graph. It is possible to construct a tree for a type expression, with inner nodes for type constructors and leaves for basic type, type name, and type variables.

The type-checker must be therefore extended, to consider also preprocessor conditioned variables declarations and their use inside expressions. The type checker must also address particular situations in which variables are never defined in some configurations, but also used in the program construct because they erroneously have not been considered by preprocessor condition, or cases in which variables are declared with a specific type for specific configuration but used as a different type (i.e., a language construct has been erroneously reached).

3.5 Impact of Preprocessor Conditionals

In this section the impact of preprocessor conditionals on variable declaration is analyzed and discussed, demonstrating the potential usefulness of the approach. Seventeen open source software systems were analyzed, most of which also studied
in [3]. Two further systems, mozilla and the Linux kernel 2.4.18, for which preprocessor conditionals play a fundamental role, were also analyzed. It was found that most of the configuration-dependent declarations were contained in the header files, and therefore the analysis was restricted on that portion of code. As discussed in Section 3.2, preprocessor directives used to prevent circular inclusion were skipped.

Figure 3.7 reports the percentage of preprocessor conditioned declaration in the software systems analyzed. In most cases, the percentage is not negligible, and is around 20% (i.e., m4, gzip, gnuplot, ghostview, gawk, emacs, bison). There are cases, which should be analyzed in more closely where the percentage is close to 40% (e.g., perl, mozilla, gs,gnuplot and the Linux Kernel).

![Graph showing percentage of preprocessor-conditioned declarations.](image)

**Figure 3.7:** Percentage of preprocessor-conditioned declarations.

High percentages for mozilla (some examples are shown in Figure 3.9) are due to factors such as:

- Handling platform-dependent code; and
- Verifying if the configuration under which the system will be compiled includes certain libraries and components.

As highlighted in Figure 3.9, complex and nested conditions should be handled: this enforces the need for the expression simplifier explained in Section 3.4.2. Similarly, analyzing perl, it was found that preprocessor conditionals are targeted to compile the interpreter on different platforms and to check if some perl add-ons and libraries are included in the configuration to be built.

![Preprocessor-conditioned declarations in the Linux Kernel subsystems.](image)

**Figure 3.8:** Preprocessor-conditioned declarations in the Linux Kernel subsystems.

A more focused analysis was performed on the Linux Kernel. The objective was to analyze the impact of preprocessor conditioned declarations on the different Kernel subsystems. To perform an analysis on different subsystems, the first-level depth subdirectory decomposition also adopted for clone-detection in Chapter II was used. In other words, a separate analysis was performed on general header files contained in the `include` directory, and then those of the six major subsystems (`ipc, fs, arch, net, drivers, scripts`). Results are reported in Figure 3.8, together with the overall results (the *Total* bar) from Figure 3.7. The percentage in the `include` directory was significant, due to the fact that most of the Linux Kernel code is customized during pre-compiling configuration (i.e., establishing the architecture,
what drivers to include, etc.). As expected, hardware-specific code (drivers and arch subsystems) exhibit a considerable percentage, especially more independent subsystems, such as net, fs and ipc.

```c
#ifdef XP_WIN
#ifdef XP_WIN16
    #if defined (XP_WIN) || defined (XP_OS2)
    #ifdef XP_MAC
    #ifdef XP_UNIX
    #if !defined(XP_RANDOM) ||
        !defined(XP_SRANDOM)
    #if defined(UNIXWARE) ||
        defined(_INCLUDE_HPUX_SOURCE) ||
        (defined(_sun) && defined(__svr4__)) || defined(SNI) || defined(NCR)
#endif
#endif
#endif
#endif
```

Figure 3.9: Preprocessor conditionals in mozilla code.

### 3.6 Conclusions

This chapter presented an approach and the architecture of a tool devoted to the automatic detection of faults caused by wrong use of configuration-dependent variables. This was obtained building an enhanced symbol table, in which the type of a variable depends on a Boolean expression obtained composing and then simplifying preprocessor conditionals dominating the variable itself. This will allow a more effective type-checking with respect to a traditional compiler, since the latter needs to compile the system for each individual configuration to be tested, while the former can automatically check expressions present in the code for all the possible configurations. Possible impact of the tool was analyzed computing the percentage of preprocessor conditioned variables present in 17 different software systems, and analyzing the purpose of these conditionals.
Bibliography


CHAPTER IV

Design Pattern Extraction

4.1 Introduction

OO design patterns represent solutions to common design problems in a given context. The most well-known OO design patterns collection is contained in the book of Gamma et al. [1]; 23 design patterns were collected and documented by the authors who also presented pattern implementation in Smalltalk and C++. Other design patterns are described in [2] and, using the design pattern description explained in [1], any design pattern catalog can be easily extended adding new patterns.

This chapter describes an approach for OO design pattern recovery firstly introduced in [3, 4], then extended in [5], where revised design pattern representation and design pattern recovery constraints are presented, as well as new case study results. The approach relies on a conservative multi-stage reduction strategy based on software metrics to extract OO design pattern candidates from software artifacts, namely the design class diagram or the source code.

While forward engineering the benefit of using design patterns is clear [6], from a program comprehension and maintenance perspective a pattern provides knowledge about the role of each class within the pattern, the reason for certain relationships among pattern constituents and/or the remaining parts of a system. In other words, the discovery of patterns in a software artifact highlights rationale of the adopted solution, representing a step in the program comprehension process and improving documentation. Consequently, in maintenance, the identification of design pattern
instances provides insight on software artifact structure and reveals places where changes, reuse, or extensions are expected. Moreover a system, which has been designed using well-known, documented and accepted design patterns, is also likely to exhibit good properties such as modularity, separation of concerns, reusability and ease of extension. Thus, design patterns can also give some indications to managers about the quality of the overall system.

The presence of patterns in a design should also be reflected in the corresponding code: the extraction of pattern information from both design and code is fundamental in identifying traceability links between different documents, explaining the rationale of the chosen solution in a given system and thus simplifying the activity of building its conceptual model.

In the approach presented here, software metrics play a central role: a design pattern is represented as a tuple of classes and relations among classes. When examining potential pattern instances, OO software metrics [7, 8] are used to avoid combinatorial explosion in checking all possible class combinations while determining pattern constituents’ candidate sets. Pattern structure is then considered to further reduce the search space. In fact, a pattern can be conceived as a graph in which nodes are classes and edges correspond to relations. Once a pattern element is chosen, remaining classes are constrained to it by the number of in-between relations. Thus, the problem of further reducing the search space is mapped into a shortest path problem where the pattern imposes the existence of a certain number of edges between candidate classes. Shortest path filtering effectively reduces the number of candidate tuples determined by the first application of software metrics. On these reduced sets, exact structural design pattern constraints are applied.

To ensure independence of the approach from the programming language and from the adopted CASE (Computer Aided Software Engineering) tools, software artifacts, code or design, are mapped into an intermediate representation, called AOL [9]. Other pattern recovery approaches [10] use language dependent representation to express different high-level structural relationships among classes. AOL is basically focused on class diagram concepts: classes, methods, attributes and their
properties, as well as relations among classes, are modeled. Software metrics, as well as structural properties, are extracted from an AST produced by parsing the AOL software artifact representation.

Since necessary conditions are applied, the proposed approach is conservative, thus if a pattern is in the code it is surely reported in the results. Consequently, sometimes spurious patterns may be reported: to reduce such false positives, design pattern constraints in terms of method delegation have been exploited. Method delegation means that a class implements one operation by simply calling an operation of another class with which it is associated, thus delegating the responsibility to it.

The chapter is structured as follows: after a review of related work in the field of design-patterns recovery and an introduction on pattern concepts, the recovery process and the pattern matching approach are presented in Section 4.4 and 4.5. Finally, experimental results obtained on industrial and public domain software are reported and discussed in Section 4.7, before conclusions.

4.2 Related Work

Design patterns extraction is a relatively young field; few works in program understanding and reverse engineering have addressed design pattern recovery. Kramer and Prechelt [11] have proposed an approach and developed a system, called Pat, that localizes instances of structural design patterns by means of structural information. It relies on the reverse engineering capability of a CASE tool repository to extract design information and uses Prolog facts to represent it and rules to express patterns. A Prolog query searches the fact database for all pattern instances. Experimental results on industrial and public domain software were provided. However, a direct comparison with the approach proposed in this chapter is not possible. The CASE tool does not pre-process source code; furthermore, Pat is case insensitive, hence it retrieves design patterns that do not really exist in the code [12].

Keller [13, 14] have developed an approach and a prototype tool for recovering and visualizing both generic and ad-hoc design patterns, given the reverse-engineered source code of a system. The tool supports recovery of design patterns using auto-
matic, manual and semi-automatic design clustering techniques. Knowledge about the existence of design patterns is preserved in a central repository. Patterns are visualized directly in the reverse-engineered source code models through three types of diagrams: the pattern-enhanced class diagram, the pattern-analysis diagram and the pattern-collaboration diagram. Quantitative data on the proposed approach are not available. The tool, at the present, privileges manual/semi-automatic recovery process and design pattern visualization.

Shull, Melo and Basili [15] have developed an inductive method to help discover custom, domain-specific design patterns in existing OO software systems. The method, however, is performed manually, although it could be greatly assisted by tools. Antoniol and Tonella [16] proposed an approach for the inference of recurrent design patterns directly from the code or the design. No assumption is made on the availability of any pattern library, and the concept analysis algorithm was adapted for the pattern inference. Seemann and Wolff [17] presented a pattern-based design recovery approach relying on structural information and class-exchanged messages.

Different approaches [18, 19], not specifically oriented towards the OO paradigm and design patterns, exploited software metrics to automatically detect design concepts and function clones in large software systems written in procedural languages.

A process, oriented to analyze software design decisions was then discussed in [20]. The role of design patterns in quality-driven reengineering was discussed in [21]. In [22] a tool to support iterations and incremental changes during the pattern recovery process was proposed. Recently, a work on the recovery of behavioral patterns has been proposed in [23]. For these patterns, a static analysis of the source code or design document structure does not suffice. To extract behavioral information, most of the approaches combine static information with dynamic information, i.e., execution traces.

### 4.3 Design Patterns

The current use of the term *pattern* is derived from the writings of the architect Christopher Alexander who has written several books on the topic [24]; Alexander’s
books describe architecture and urban planning. However, the key idea is applicable to many other disciplines including software development. Software patterns became popular with the wide acceptance of the book by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [1]. Since the early work of E. Gamma [25], there has been a flourish of interest and activities around patterns and pattern languages [26, 27, 28, 29].

Beyond the design phase, patterns have been proposed for almost each phase of the development process [27]: requirement patterns, analysis patterns, architectural patterns, design patterns and (code) idioms represent at different abstraction levels proven solutions to recurring problems within certain contexts (another good source of patterns is the Pattern Home Page [30].

![Bridge design pattern diagram](image)

Figure 4.1: Bridge design pattern.

This work is focused on patterns at design level: such patterns correspond to well-known and frequently reused design micro-architectures, excluding other higher or lower levels category of patterns. A design pattern description encompasses its static structure, in terms of classes and objects participating in the pattern and their relationships, but also the intent and the participants’ exchanged messages, i.e., the pattern behavior. The description of the solution tries to capture the essential insight which the pattern embodies so that others may learn from it and make use of it in similar situations: patterns help to create a shared language for communicating insight and experience about these problems and their solutions.

According to the taxonomy proposed by Gamma, design patterns can be classified into *creational, structural* and *behavioral*. Creational patterns concern object
creation, structural patterns capture classes or object composition and behavioral patterns deal with the way in which classes and objects distribute responsibility and interact.

![Diagram of the Adapter design pattern.](image)

**Figure 4.2**: Adapter design pattern.

From a program comprehension and reverse engineering perspective, the complexity of extracting information from a design or source code is not the same for the different pattern categories. While for structural design patterns information is explicit in their syntactic representation, for the other two pattern families behavioral information must be recovered, which might involve the analysis of messages exchanged and the code of class methods. This also implies that the recovery of creational and behavioral patterns is not always feasible at design stage, where in most cases the only information available is the bare static information obtainable from class diagrams. Even the class diagram information could be poor: it is not unusual that neither data member type nor method signature is specified. Moreover, almost always, no information is available on method body, neither as pseudo-code nor as the set of messages sent to/received from other objects.

For these reasons, this work focused on the recovery of five of the seven structural design patterns proposed in [1]: the adapter, the proxy, the composite, the bridge and the decorator. The remaining two structural patterns (the facade and flyweight) were not considered since their structure does not provide enough information to retrieve their instances. In particular, the facade pattern consists of a single class...
while the flyweight pattern has much of its information embedded in the methods and attributes of its participants. Identifying flyweight instances would require identifying a FlyweightFactory, creating and managing flyweights and distinguishing flyweight classes storing intrinsic and extrinsic states.

Focusing on structural design patterns does not mean that non-structural design patterns cannot be represented and recognized: it simply means that the extraction process does not exploit behavioral information, thus a much higher number of false positives can occur with respect to the structural design pattern category.

In Figures 4.1 and 4.2, the UML [31] class diagrams of the Bridge and the Adapter patterns are depicted. The Bridge design pattern package is a common technique used to provide flexible and extensible implementations when an abstraction can have several implementations. For example, when implementing a portable Window abstraction in a user interface toolkit [1], by putting the abstraction and the implementation in separate hierarchies, the abstraction and the platform-specific implementation are decoupled. Clients create an Abstraction which is not bound to any ConcreteImplementor, and it is the responsibility of the Abstraction to get the proper Implementor.

In the Adapter pattern, the Adapter object adapts the interface exported by the Adaptee object, so that the Adaptee services can be called with the different calling conventions of the Target object. This implies that a subclass of Target, the Adapter is created, which actually implements the abstract operation Operation exported by Target, by delegating the task to the SpecificOperation of Adaptee.

4.4 Recovery Process

The approach relies on the design pattern recovery process represented in Figure 4.3. Both design and source code can be the input of the recovery process. The process consists of the following activities:

1. AOL Representation Extraction: in this phase an AOL representation is recovered from design or code through respectively a CASE2AOL or a Code2AOL
2. **Pattern Recognition**: a multi-stage **Constraint Evaluator** process localizes design pattern instances on decorated AOL ASTs;

3. **User Interaction**: the recognition process, extracted metrics and pattern results visualization are implemented using HTML pages displayed by a web browser.

The pattern recovery process is based on AOL, on software metrics computed on AOL ASTs and on a design pattern library. The following subsections will highlight the key issues of these elements and the related implications.

### 4.4.1 Design Pattern Library

The pattern recovery process relies on a design pattern library, thus there are similarities with the program comprehension and architectural recovery approaches based on *cliché* matching and plan recognition. The proposed approach relies on a strategy based on design local properties, represented by means of software metrics, to avoid the design pattern recovery process complexity combinatorial explosion. However, an instance of design pattern may appear in several different forms due to variations: this work shares with the program understanding and architectural recovery community the problems identified by Wills [32]. Wills classifies the main
sources of variation as: *syntactic variation, implementation variation, delocalization, organization variation, redundancy, unrecognizable code* and *function sharing.*

*Syntactic variation* mostly regards the syntactic level clichés: they are recognized directly with pattern matching on the AST. The different cliché forms corresponding to the different language constructs that may be used in specifying a cliché are codified as different *implementations* [32] of the same concepts. A cliché recognizer embodies the knowledge of all the different forms that a certain cliché can assume. In the design pattern recovery domain, syntactic variability is mostly related to the different ways in which an aggregation can be implemented i.e., by means of pointers or array data members.

*Implementation variation* is related to the fact that a given concept may be implemented in different ways: an aggregation may be implemented with a list or a set or any other user defined type. Another example is the depth of the inheritance tree between a superclass and a derived class participating in a pattern (see for example the *Bridge*) it may or may not be exactly one level.

As for syntactic variation, also in this case the different cliché forms are codified as different implementations (i.e. a set of alternative forms) of the same concepts. In other words, a preliminary work is required to enrich a standard design pattern library with the company/ project expected implementation variations.

*Delocalization* [32, 33, 34] and *organization variation* are quite similar issues. They occur when the components of a given cliché are not contiguous but spread over different files in the source code. This phenomenon could cause some problems to a purely syntactically-based cliché recognition. By considering the entire system design we are guaranteed that if an instance of a design pattern is present it will be surely reported.

The *redundancy* problem occurs when a part of a cliché appears more than once in the same instance of a cliché: by applying only necessary conditions a superset of the actual design pattern is retrieved.

The *unrecognizable code* phenomenon is also called *partial recognition*: the recognition system, because of the incompleteness of its cliché library, is not able to classify
every line of the source code or class in a design as belonging to some cliché. What happens actually is that a forest of partial matching is generated, and some parts of the program remain uncovered. For the design pattern recovery domain, partial recognition of a program is expected since usually most of a program is devoted to performing the computation related to its application domain, while only a small fraction of the design/code is devoted to implementing design pattern structures.

*Function sharing* is also called *optimization* [32] or *overlapping implementations* [35]. It happens when the same portion of code is shared among two or more clichés. Here the implementations of the clichés overlap. This phenomenon does not represent an actual problem given the software metrics approach, the organization of the cliché library and the recognition process. This simply means that in the recovered family of design patterns, some components will be repeated.

The five design patterns were represented in the pattern library in the [1] *canonical* forms. However, since in OO languages such as C++ an aggregation may be implemented by means of a pointer to a complex data structure, the *Bridge*, the *Composite* and the *Decorator* were represented as *soft patterns*. According to the Gamma book [1], the aggregation was substituted by an association.
4.4.2 AOL Representation and Parsing

AOL has been designed to capture OO concepts in a formalism independent from programming languages and tools. The language is a general-purpose design description language, capable of expressing concepts available at the design stage of OO software development. This language is based on the UML [31], a de facto standard in OO design. Figure 4.4 shows the AOL description of the object model correspondent to the Adapter design pattern in Figure 4.2.

The AOL specification derived either from the code or from the design is parsed by the AOL Parser, producing an AST representing the object model. The parser also resolves references to identifiers, and performs some simple consistency checking.

4.4.3 Class Metrics Extraction

Software metrics are usually exploited to characterize artifact properties. The proposed approach mainly focuses on object class specific properties, properties that can be used to avoid the complex combinatorial explosion of the design pattern recovery process. The Metrics Extractor traverses the AOL AST, decorating it with the single-class metrics used by the subsequent module, the Constraint Evaluator, which implements the multi-stage recognition process described in the Section 4.5.

Software metrics are displayed to the user at the end of the recognition process more as a by-product, since the goal of this approach is to recover design patterns.

Software metrics are the essential means to reduce the search space dimension: they have to be functional to the design pattern recovery process. It is therefore worth focusing only on those software metrics that can be directly derived by the design patterns structure or properties, and furthermore could be regarded as constraints to be satisfied by candidate patterns. Hence, the number of relations (aggregations, associations, inheritances) and methods are the key metrics of interest; more precisely, the metrics computed for each class are:

- Number of public, private and protected attributes;
- Number of public, private and protected operations;
• Number of direct subclasses, number of direct superclasses;

• Number of association and aggregation relations in which a class is involved; and

• Total number of attributes, methods and relations.

The set of the above metrics suffices to perform recognition process for the structural design patterns as described in this chapter.

Other metrics, such as depth of inheritance tree or number of derived (directly or not) classes can be easily computed and could be used to recover different kinds of patterns or to augment the precision of the recovery process. For example, \textit{facade} instances could be identified using class relation \textit{fan-in} and \textit{fan-out} metrics, searching for classes with high fan-in and fan-out. However, the approach is not guaranteed to be conservative, since the fan-in and fan-out values are not constrained by the \textit{facade} pattern representation: applying thresholds on these values could result in missing some true pattern instances.

4.5 Pattern Recognition

Finding an instance of a design pattern involves identifying a set of classes that exhibit the exact pattern properties: relationships, behavior and intent. Structure and delegations among classes must hold, behavior in terms of class responsibility distribution and exchanged messages must be checked.

Furthermore, the pattern intent should be elicited. For example, the key difference between \textit{Adapter} and \textit{Bridge} lies in the intent. The \textit{Adapter} is a wrapper, it resolves an incompatibility between already existing interfaces. The \textit{Bridge} decouples an abstraction from its possibly numerous implementations. Both \textit{Adapter} and \textit{Bridge} may accommodate different implementations; however, the \textit{Bridge} makes objects work together before they are implemented; the \textit{Bridge} was intentionally designed, e.g., to accommodate new implementations as the system evolves. The \textit{Adapter} may be useful to avoid rewriting code because it makes things work together after they have been designed.
Intent redocumentation requires discussion with developers or, whenever this is not possible (e.g., dealing with public domain code) at least manually browsing the code attempting to discover the intent of the developers. This last activity unfortunately may introduce a certain degree of subjectivity: it must be decided whether or not a certain structure was generated by chance or represents the solution to the recurring problems for which the design pattern was invented.

Finally, the matter is further complicated whenever a class may participate in multiple design patterns; for instance a Bridge and an Adapter may be partially overlapped. In the absence of reliable documentation, during the manual verification it was safely assumed that both design patterns could possibly be present. Moreover, whenever they were judged compatible with the pattern definition they were counted as true design patterns regardless of being overlapped.

![Diagram](image)

**Figure 4.5:** Constraint Evaluator multi-stage filtering.

A pattern may be represented as a tuple \(<e_1, \ldots, e_k>\) of elements \(e_i\) and a set of properties among them. The representation of properties may depend on the property nature: structural properties may be represented by the class diagram or other equivalent textual notations, behavioral properties may be expressed by interaction diagrams or state diagrams, intention may be described by a plain text. As already discussed, this work focuses the attention on structural design patterns, exploited from the set of structural relations, \(\mathcal{R}\), which almost entirely characterize structural design patterns. Given a pattern \(p = (<e_1, \ldots, e_k>, \mathcal{R})\) with cardinality \(k\), a brute force approach to identify all possible pattern candidates in a design containing \(n\) classes would require computation of all the dispositions without repetition of the design classes \(k\) by \(k\) (i.e., \(n(n-1)\ldots(n-k+2)(n-k+1)\)) and a check of the validity of the \(\mathcal{R}\) relations for each of them. The resulting worst-case complexity is
therefore $O(n^k)$.

To reduce the complexity, the multi-stage recognition process shown in Figure 4.5 has been adopted. Each class in a pattern candidate must exhibit a number of aggregations, associations and inheritances consistent with the given pattern prototype. Hence a first search space reduction can be accomplished by means of Class Level Metrics Constraints. The output of the first block in Figure 4.5 is a set of candidate classes for each pattern constituent: classes which exhibit a compatible number of relations with respect to the design pattern searched.

If the inheritance, the aggregation and the association are not distinguished, a design pattern may be thought of as a graph: nodes represent classes and edges represent relations. By disregarding the specific type of traversed edges, each couple of pattern constituents may be characterized by the minimum number of in-between relations: the second stage of Figure 4.5 constructs reduced candidate classes filtering out from candidate classes sets those couple elements, belonging to distinct candidate classes, not satisfying the Shortest Path Constraints.

In the last structural filtering stage, reduced candidate classes elements are required to verify the exact correspondence with the pattern prototype relations. This step, highlighted in Figure 4.5 as Structural Constraints, requires the explicit construction of all possible tuples; however, the tuple elements are extracted from reduced candidate classes and only a subset of the $O(n^k)$ combinations is actually examined.

Although in practice reduction factors of several orders of magnitude can be observed by exploiting the process described above, from a theoretical point of view it was not possible to devise the reduction factor for a generic design. In principle, a design could exist such that no reduction is accomplished by the metrics-based stages. A trivial, but meaningless example is a design in which each class is related in some way with all the other classes. Thus, the effectiveness of the metrics-based stage reduction depends on the topology of a design and it is not a-priori predictable. Section 4.7.2 will present actual values for the reduction factor on industrial and public-domain programs.
4.5.1 Class Level Metrics Constraints

Each element \( e_i \) belonging to a given design pattern \( p \) is characterized by a tuple of metrics which allows to extract, with linear complexity, a candidate set for each pattern element. In other words, let \( p = (e_1, \ldots, e_k, \mathcal{R}) \) be a pattern (with elements \( e_1, \ldots, e_k \)) belonging to a pattern collection \( \mathcal{P} \). Let \( M_p = (m_1, \ldots, m_k) \) be the tuple of metrics characterizing pattern \( p \), where each \( m_i \) is the array \( m_{ij}(\cdot), j = 1, \ldots, l \) of software metrics chosen to describe \( e_i \). For any given pattern \( p \), \( e_i \) candidate class is the subset of classes \( x \), belonging to the design \( D \), that exhibit software metric values \( m_{ij}(x) \) compatible with \( m_{ij}^j(e_i) \):

\[
C_p(i) \overset{\text{def}}{=} \{ x| x \in D \land m_{ij}(x) \triangleright m_{ij}(e_i) \} m_i \in M_p, j = 1, \ldots, l
\]

These represent necessary conditions: \( \triangleright \) was used to state that \( x \) metrics values, \( m_{ij}(x) \), are enforced by the pattern structure. To ensure admissibility, \( \triangleright \) was implemented as \( \geq \) and no upper-bound values were imposed on relations: each class was allowed to be also related with any number of other classes in the design (beyond those participating in the pattern). Clearly, the same design element \( x \) could be in more than one \( C_p(i) \). However, actual pattern instances are surely within the tuples extracted from the set collection \( C_p(1), \ldots, C_p(k) \).

As an example, suppose we want to identify instances of the Adapter pattern in the design of Figure 4.6. Classes are tagged with \( T \), \( A' \) or \( A'' \) if their metrics values are compatible with respectively the Target, Adapter or Adaptee pattern elements. The given design contains 11 classes, so in principle all the dispositions without repetition of 11 classes 3-by-3 should be checked, i.e., 990 class triples.

For simplicity’s sake the set of metrics used was limited to four of the above metrics, namely the number of direct subclasses, direct superclasses, association and aggregation relations, since they are the most important to identify Adapter instances. The Adapter structure (see Figure 4.2) imposes the following constraints: the Target must have at least one subclass, the Adapter must have at least one superclass and one association and the Adaptee must participate in at least one
association, independently of the specific classes they are related to in these relations.

Let the Adapter pattern be defined as:

\[ p = \langle \text{Target, Adapter, Adaptee} \rangle, \]
\[
\{ \text{subclass(} \text{Adapter, Target}) \}, \]
\[
\text{superclass(} \text{Target, Adapter}) \], \]
\[
\text{association(} \text{Adapter, Adaptee}) \} \}
\]

Assuming that the metrics in each array \( m_{i,j} \) represent respectively the number of subclasses, superclasses, association and aggregation relations, a triple of classes \( < c_1, c_2, c_3 > \) is an instance of the Adapter pattern only if:

\[
\#\text{subclasses}(c_1) \geq 1
\]
\[
\#\text{superclasses}(c_2) \geq 1
\]
\[
\#\text{associations}(c_2) \geq 1
\]
\[
\#\text{associations}(c_3) \geq 1
\]

Given the Figure 4.6 design, the sets \( C_{\text{Adapter}}(\text{Target}), C_{\text{Adapter}}(\text{Adapter}), C_{\text{Adapter}}(\text{Adaptee}) \) are:
\[ C_{\text{Adapter}}(\text{Target}) = \{A, C, F\} \]

\[ C_{\text{Adapter}}(\text{Adapter}) = \{B, C\} \]

\[ C_{\text{Adapter}}(\text{Adaptee}) = \{B, C, E, H\} \]

The first stage reduces the example to 24 triples, obtaining a reduction factor of 41.25.

4.5.2 Shortest Path Constraints

To further reduce the search space, let us choose the \( C_p(i) \) set with minimum number of elements; \( e_j \) candidates (elements of \( C_p(j) \) and \( i \neq j \)), are related to \( e_i \) candidates (elements of \( C_i \)) by the minimum number of \( e_i - e_j \) in-between relations. For each \( y \) in \( C_p(i) \), a restriction \( R_{p,i,j}(y) \) over the set \( C_p(j) \) is computed as:

\[ R_{p,i,j}(y) \overset{\text{def}}{=} \{x | x \in C_p(j) \land ShPath(y,x) = ShPath(e_i,e_j)\} \]

where \( ShPath(y,x) \) is the shortest path between two classes \( y \) and \( x \) in a design, measured as the number of relations traversed to reach \( x \) from \( y \), independently of the nature of the relations. Each \( y \) in \( C_p(i) \) induces pattern candidate tuples constructed over the reduced candidate sets \( R_{p,i,j}(y) \):

\[ <r_{i,1}, r_{i,2}, \ldots, y, \ldots, r_{i,k}> \mid r_{i,j} \in R_{p,i,j}(y) \mid j = 1, 2, \ldots, k \ j \neq i \]

Notice that, even if theoretically speaking, \( R_{p,i,j}(y) \) computation could require computation of the all pair shortest paths, with cubic complexity [36]; in practice, since design patterns are micro-architectures, typical values for \( ShPath(e_i,e_j) \) are limited and usually below 4. As a consequence the complexity observed in practical cases is almost linear with the size of \( C_p(i) \).

Returning to the Adapter pattern recovery example, the \( C_{\text{Adapter}}(i) \) set would be the Adapter set i.e., \( C_{\text{Adapter}}(\text{Adapter}) \), which contains two classes, B and C.
By choosing $C_{Adapter}(Adapter)$, given the Adapter pattern structure, Targets and Adaptees belong to the neighborhood of each class in the $C_{Adapter}(Adapter)$ (Figure 4.7)) reachable in one step:

$$R_{Adapter, Adapter, Target}(B) = \{A, C\}$$
$$R_{Adapter, Adapter, Adaptee}(B) = \{C\}$$

$$R_{Adapter, Adapter, Target}(C) = \{A\}$$
$$R_{Adapter, Adapter, Adaptee}(C) = \{B, E\}$$

Thus the set of all possible triples also satisfying these constraints is:

$$R = \{< A, B, C >, < A, C, B >, < A, C, E >\}$$

Notice that, for the design patterns considered in this work, a class is not allowed to play more than one role, i.e., $< C, C, B >$. The reduction factor depends on the choice of the $C_p(i)$ set. Different classes in a pattern imply different neighborhoods for computing the $R_{p,i,j}(y)$, according to the topology of the pattern and of the design analyzed. The heuristic of choosing the $C_p(i)$ set with the minimum number of elements does not assure the highest reduction factor, since such choice could lead to consideration of larger neighborhoods. A different selection criterion could be to choose the $C_p(i)$ set, which corresponds to the pattern element whose distances from the other pattern elements are minimal. However, also this criterion does not guarantee the optimality.

4.5.3 Structural Constraints

The exact design pattern structural constraints are applied to tuples computed in the previous step: $< r_{i,1}, r_{i,2}, \ldots, y, \ldots, r_{i,k} >$; the set $S$ of candidate tuples is constructed as:
Figure 4.7: Neighborhood of classes B and C (respectively indicated with solid and dashed lines) at distance 1 on the design graph.

\[
S \overset{\text{def}}{=} \{ z | z = \langle r_{i,1}, r_{i,2}, \ldots, y, \ldots, r_{i,k} > \land \\
\forall \mathcal{T} \in \mathcal{R} : \mathcal{T}(e_q, e_s, \ldots e_t) \Rightarrow \mathcal{T}(r_{i,q}, r_{i,s}, \ldots, r_{i,t}) \}
\]

where \(q, s\) and \(t\) are generic indices of a subset of pattern elements that participates in a given relation \(\mathcal{T} \in \mathcal{R}\).

With respect to the Adapter pattern recovery example, imposing the exact structural relation constraints in terms of inheritance, aggregation and association does not remove any candidate. However, from Figure 4.2, another structural constraint must hold: Target and Adapter classes must have at least one operation, with the same name, which may be abstract in the Target and is implemented by the Adapter.

Thus, with reference to Figure 4.8, class B cannot play the Adapter role since it does not have common operations with Target. The resulting set \(S\) of tuples satisfying also the structural constraints for the Adapter pattern search is thus:

\[
S = \{ \langle A, C, B \rangle, \langle A, C, E \rangle \}
\]
Only structural properties were applied: the above constraint requires the Adapter candidate to override a Target method; that is, there must exist at least a Target method implemented by the Adapter. The semantic of the overridden method is not considered, it cannot be guaranteed that the overridden methods implement a wrapper.

4.5.4 Delegation Constraints

When detailed information on methods call is available, the set $S$ is taken as input of the last stage in which the delegation constraints are checked and the final set tuples produced. Delegation constraints can be represented as proposed in [16]. For example, let us assume that a design pattern requires that $e_i$ invokes method $M$ of $e_j$. This can be represented by the relation $(i,j)_{\text{call}(M)}$, where $c(M)$ is a label for the call to method $M$. If the invocation is associated with a specific method from a class – rather than the class as a whole – the label can be modified into $\text{call}(M_1, M_2)$ to indicate that method $M_1$ from the first class calls method $M_2$ from the second class. For example, the Adapter constraint can be expressed as: \[(\text{Adapter, Adaptee})_{\text{call}(\text{Adapter->op()}, \text{Adaptee->op()})}\].

The Code2AOL Extractor is able to safely identify superset of methods invoked by a given method; verifying the presence of an actual method delegation allows the elimination of discovered patterns that satisfy only structural requirements. For example, consider the Adapter design pattern in Figure 4.2. Once Target, Adapter, Adaptee candidates have been identified, we further impose the delegation constraint: the Adapter must issue a call from one of his methods to an Adaptee’s method.

If we assume that method calls are those represented in Figure 4.8, then only one triple, $<A, C, E>$ may be an Adapter instance, since C calls the op method exported by E, while B does not call any methods of C, and vice-versa.

The final set of Adapter pattern candidates is thus:

$$P = \{<A, C, E>\}$$
Figure 4.8: Design example with method call information for delegation-based stage.

Having applied only necessary structural and delegation conditions, it is not assured that the semantics of $< A, C, E >$ is actually consistent with an Adapter: designer intention, classes behavior could differ from the Adapter design pattern; nevertheless, the instance found is possibly the only Adapter instance in the given design.

4.6 Tool Support

The process for extracting candidate design patterns shown in Figure 4.3 has been completely automated. In particular, the recovery process relies on the following tools that extract the AOL from design and from code respectively:

- A CASE2AOL Extractor module has been implemented for the StP/OMT (Object Modeling Technique) CASE tool (StP Manuals) to obtain an AOL specification of internal object models from its repository.

- a Code2AOL Extractor module has been developed and works for the C++ language.

Extracting information about class relationships from code may be more difficult than from design, and the result might have some degree of imprecision. In fact, there are intrinsic ambiguities, given two or more classes and a relation among them, due to
the choice left to programmers implementing OO design. Associations can be instantiated in C++ by means of pointer data members or by inheritance. Furthermore, aggregation relations could result either from templates (e.g., \texttt{list<tree>}), arrays (e.g., \texttt{Heap a[MAX]}) or pointers data member (e.g., \texttt{Edges=new GraphEdge[MAX]}). In the present work, an aggregation is recognized from code if and only if a template, an object array, or an instance of an object is declared as data member. All the remaining cases (i.e., object pointers and references both as data members and formal parameters to methods) give origin to associations.

Since the above choice may be over-restrictive, it does not take into account the \textit{syntactic variation} and \textit{implementation variation} identified by Wills [32]; structural design patterns containing aggregations have been represented in two forms: the canonical and a more flexible one (referred to as \textit{soft} version) in which aggregations are substituted by associations, since an aggregation is actually a special form of association.

To perform design patterns extraction, the following tools have been implemented using the \textit{Perl} programming language:

\begin{itemize}
  \item An \textit{AOL Parser};
  \item A \textit{Metrics Extractor}; and
  \item A \textit{Constraint Evaluator}.
\end{itemize}

\section{Case Studies}

\subsection{Case Study Description}

The extraction process described in the previous sections has been applied on public-domain and industrial software. There are some advantages in analyzing public-domain software: source code can be easily obtained and results can be compared with those of related works. On the other hand, it is quite difficult to find a public-domain software having both code and design available. As a consequence, the recovery process was applied both to public-domain and industrial code but only on industrial design.
LEDA (version 3.4) and libg++ (version 2.7.2) are two well-known C++ libraries of foundation classes. The first, Library of Efficient Data types and Algorithms, was developed and is distributed by Max-Planck-Institut für Informatik, Saarbrücken, Germany; the second is part of the GNU Free Software Foundation C++ development environment.

galib (release 2.4) is a C++ GA library to solve optimization problems, developed at the Massachusetts Institute of Technology. mec (release 0.3) is a trace-and-replay program developed by Michael Chastain. mec-trace runs a target program and traces it. socket (release 1.10) is a library for inter-process communication developed by Gnanasekaran Swaminathan at the University of Virginia.

To assess the use of design patterns in an industrial environment and to verify consistency between code and design, a second experiment was conducted on design and code of industrial software for telecommunications. Eight components of a complete system were analyzed. All components were documented with OMT design and were developed in the same environment by the same teams using the same language (C++).

As shown in Table 4.7.1 and Table 4.7.1, software analyzed in both experiments have sizes and other OO characteristics spread fairly evenly across a broad range.

<table>
<thead>
<tr>
<th></th>
<th>galib</th>
<th>LEDA</th>
<th>libg++</th>
<th>mec</th>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCs</td>
<td>20507</td>
<td>11582</td>
<td>40119</td>
<td>21006</td>
<td>3078</td>
</tr>
<tr>
<td>Classes</td>
<td>55</td>
<td>208</td>
<td>167</td>
<td>32</td>
<td>29</td>
</tr>
<tr>
<td>Attributes</td>
<td>206</td>
<td>426</td>
<td>308</td>
<td>94</td>
<td>17</td>
</tr>
<tr>
<td>Operations</td>
<td>916</td>
<td>4610</td>
<td>2863</td>
<td>316</td>
<td>288</td>
</tr>
<tr>
<td>Aggregations</td>
<td>10</td>
<td>166</td>
<td>20</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>Associations</td>
<td>97</td>
<td>334</td>
<td>91</td>
<td>103</td>
<td>23</td>
</tr>
<tr>
<td>Inheritances</td>
<td>36</td>
<td>85</td>
<td>95</td>
<td>6</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 4.1: Public-domain code characteristics.

It can be argued that very few instances of design patterns were retrieved. This happens because the analyzed systems were not conceived taking into consideration design patterns. To verify this conjecture design patterns have been analyzed and extracted from ET++: the software system that inspired the Gamma, Helm, Johnson
and Vlissides book [1]. ET++ is an OO framework containing basic data structure, support for input/output and user interface building blocks; it evolved from the initial implementation to the currently available 3.0 release which can be downloaded from ftp.ubilab.ubs.ch. ET++ release 3.0 contains about 100 KLOCs of C++ code, with 704 classes; due to software evolution many changes have been made since early ET++ release described in [1]: design pattern syntactic variation, implementation variation as well as other software changes took place. For example, the stream class hierarchy changed; StreamDecorator, ASCII7Stream, CompressingStream no longer exist; moreover the Stream class is not part of any of the derived classes; so the decorator is no longer there.

4.7.2 Case Study Results

This section reports and discusses results obtained analyzing the systems described in the previous subsection. The most commonly used measures of retrieval effectiveness are recall [37] and precision[37]. Recall is the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the “database”. Precision is the ratio of the number of relevant documents retrieved over the total number of documents retrieved.

Although the number of documents in the “database” (i.e., design patterns present in a given system) is not known a-priori, since the approach being conservative, we need only to assess the number of relevant documents retrieved for a given query (i.e., number of true design patterns recognized). Precision is then computed as the
ratio of true design patterns over the number of retrieved pattern candidates.

For both the public-domain and industrial software, the verification of the actual pattern instances present in the code has been done manually, starting from the results of the extraction process and checking which of the identified patterns was an actual pattern. In the present work, each time a doubt on a pattern instantiation arose, the book [1] was considered as the reference in deciding whether or not that instantiation was actually representing a design pattern. This process took us about 3 hours, which is reasonable being that the total size of the analyzed code was about 400 KLOCs.

<table>
<thead>
<tr>
<th>System</th>
<th>Initial Metrics</th>
<th>Shortest Path and Structural Constraints</th>
<th>Delegation Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>galib</td>
<td>157410</td>
<td>1018</td>
<td>23</td>
</tr>
<tr>
<td>LEDA</td>
<td>6434670</td>
<td>2545</td>
<td>14</td>
</tr>
<tr>
<td>libg++</td>
<td>4410780</td>
<td>488</td>
<td>12</td>
</tr>
<tr>
<td>mec</td>
<td>243600</td>
<td>312</td>
<td>30</td>
</tr>
<tr>
<td>socket</td>
<td>21924</td>
<td>269</td>
<td>4</td>
</tr>
<tr>
<td>Avg Ratio</td>
<td>2388</td>
<td>55.7</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Table 4.3: Reduction of candidates through the stage filters for the Adapter design pattern.

In a conservative approach, as presented in this chapter, all pattern instances present in a software artifact are retrieved: hence perfect recall is obtained. However, recall and precision of the recovery process depends on the exactness of the AOL design input. When the AOL representation is extracted from design it faithfully describes the corresponding design information, while, when extracted from code, associations could be recognized instead of aggregations. By introducing soft patterns, we are guaranteed that we do not miss pattern instances due to aggregation and association misclassifications. Of course, more pattern instances are extracted: the approach guarantees a 100% recall at a price of lower precision values. Renouncing the admissibility could enhance precision but would also cause generation of false negatives. The desired trade-off between precision and recall depends mostly on the user objectives: for some tasks a perfect recall might be preferable because the user does not want to miss any of the actual instances, while for other tasks a higher precision would be preferred since the cost of discriminating relevant items from false
positives could be high.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>galib</th>
<th>LEDA</th>
<th>libg++</th>
<th>mec</th>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ND</td>
<td>D</td>
<td>T</td>
<td>ND</td>
<td>D</td>
</tr>
<tr>
<td>Adapter</td>
<td>23</td>
<td>9</td>
<td>4</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>Bridge</td>
<td>13</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Proxy</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>33</td>
<td>62</td>
<td>-</td>
<td>113</td>
<td>694</td>
</tr>
<tr>
<td>Precision (%)</td>
<td>3.7</td>
<td>44.4</td>
<td>-</td>
<td>11.1</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4.4: Results of pattern instances recovery on public-domain code.

Precision and precision ratios between different stages allow to assess both the overall system and single stage effectiveness. Table 4.7.2 shows the number of pattern candidates after each of the stage filters for the analyzed public-domain system, with respect to the adapter design pattern. Notice that, since no actual design patterns were identified in socket, the socket package was removed from precision evaluation and is not reported in Table 4.7.2 and successive tables.

Similar results were obtained for the other patterns considered in this work. The last row gives the average ratio of the patterns retrieved by two subsequent stages. An effective reduction of several orders of magnitude can actually be observed for each filtering stage. The first filter, class-level metrics plus shortest path, reduces the input by three to four orders of magnitude, showing the effectiveness of the use of metrics to prune the search space. The structural filter reduces by one to two orders of magnitude, depending on the specific pattern, while the delegation filter reduces the input by two to three times.

<table>
<thead>
<tr>
<th>pattern</th>
<th>Sys3</th>
<th>Sys4</th>
<th>Sys5</th>
<th>Sys7</th>
<th>Sys8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>Bridge</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BridgeSoft</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.5: Results of pattern instances recovery on industrial design.

Table 4.7.2 gives the number of pattern instances retrieved for each system in the public domain test suite for each structural design pattern. Columns labeled with ND report values computed without using the delegation constraint, those labeled
with D using the delegation constraint. T columns report the patterns manually
classified as true patterns. It can be observed that the most frequently found pattern
is the adapter. In this approach the aggregation constraint was relaxed in order to
find composite and decorator patterns. In other words, the composite and decorator
patterns found contained are compositeSoft and decoratorSoft.

The last Table 4.7.2 row gives the precision values of the pattern extraction
process, results are showed with and without the delegation constraint. Precision is
computed over all patterns, by summing the numbers of each column, ND, D and
T, and then computing $T/ND$ and $T/D$. As clearly shown by the table, by also
imposing the delegation constraint, we obtain a 60% average increase in precision,
at the expense of a limited increase in execution times.

As regards the industrial system, both design and code have been analyzed. De-
sign information has been recovered from the corporate database using the CASE2AOL
Translator developed for StP/OMT. Delegation constraint could not be checked on
design because the information about the methods called by each method at design
stage was not available. Therefore, in Table 4.7.2 the retrieved patterns correspond
to the output of the structural constraint evaluation stage. Design patterns were
not retrieved in several components, neither in the design nor in the code: patterns
seem seldom used. Table 4.7.2 does not include the component for which no pattern
instance has been found.

A comparison of the pattern instances retrieved on the design and those retrieved
on code shows that there is no intersection between these two sets; that is, all the
patterns discovered in the design are not actually present in the code. This can
be partially explained for three reasons: first, when working on design, there is no
information available about delegation, therefore more patterns than those actually
present tend to be found. Second, code often includes a collection of classes reused
from libraries or COTS that are not modeled in the design. Third, design documents
are often not completely consistent with the code, in that after code modifications,
they are not properly updated to reflect the changes, hence the gap between design
and code may become relevant.
<table>
<thead>
<tr>
<th>Pattern</th>
<th>ND</th>
<th>D</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>1944</td>
<td>290</td>
<td>160</td>
</tr>
<tr>
<td>Bridge</td>
<td>207</td>
<td>159</td>
<td>90</td>
</tr>
<tr>
<td>Proxy</td>
<td>93</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Composite</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>51</td>
<td>26</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 4.6: Structural design pattern recovery on ET++.  

### 4.7.2.1 A Pattern-Based System: ET++

As Table 4.7.2 clearly shows, ET++ contains a very high number of pattern candidates and a relevant work was required to manually verify them. ET++ contains instances of multiple hits: 12 Bridges were overlapped with Adapters and a Proxy was overlapped with a Decorator. The overlapping phenomenon and the patterns were not documented in [1]; out of 12, 7 were classified as implementing a Bridge overlapped to an Adapter, and both the Decorator and the Proxy were judged as true design patterns. The classification was to some extent subjective: three expert OO developers were asked to classify and reach a consensus on the pattern candidates. Given the low number of multiple hits, the precision result is only slightly affected.

### 4.8 Conclusions

An approach and a multi-stage process to extract structural design patterns from object oriented artifacts, design or code, has been presented. With respect to related work [11, 13], the proposed approach works on design and code which are both expressed in AOL, exploits software metrics to reduce search space complexity and makes use of method delegation information to further reduce the cardinality of the set of the retrieved pattern candidates and augment retrieval precision.

Experimentation shows that OO software metrics are extremely helpful to reduce the problem search space allowing to achieve acceptable computation times. The first step of the proposed process relies entirely on a family of well-known and widely used OO software metrics. Without using the metrics-based filter, computational times rapidly become unacceptable as the system size grows from small to medium.

Exact structural constraints are then exploited to further reduce the candidate
set and finally, by means of the method delegation filter, precision is remarkably enhanced. As demonstrated from the results, the delegation-based filter effectively improves system performance, proving that the existence of an association between two classes, most of the time, is not enough to ensure that an actual method call is issued between the two classes as the given pattern may require.

Experiments have been performed on public-domain code and on industrial code to assess the effectiveness of the approach. On both industrial and public-domain code, the approach performs quite well in terms of computational complexity and retrieved patterns, showing on public-domain code an average precision of 81%, and an increase using delegation of about 60% with respect to the use of structural constraints alone. In general very few patterns were found even if the systems were not designed explicitly using design patterns. On average, on public-domain applications, an average of 20 patterns every 100 KLOCs was found, while the patterns actually present were 12 every 100 KLOCs. On industrial code, the number of patterns actually present in the code was so small (2 patterns for 130 KLOCs), a comparison was difficult. It was observed that patterns retrieved from design and code had no intersection, that is, the design information was not consistent with code as regards pattern traceability. On the contrary, ET++ contains about 280 design patterns per 100 KLOCs; such a high discrepancy does not have any explanation except for the totally different application (a framework), programmer culture and skill.

Results produced can be useful in program comprehension and maintenance in several ways. Localizing design pattern instances in a design provides explicit knowledge about each class and object participant, their interactions and their underlying intent. Finding adapter pattern instances (for example, signal classes (the Adaptee) which are used in multiple contexts requiring different interfaces) at the same time proxy patterns indicates the existence of large objects or remote objects that are accessed through a surrogate object. Bridge patterns decouple abstractions from their implementations so they show program points in which change or reuse is expected. The knowledge of such information facilitates the task of a maintenance programmer, who is analyzing a program’s design, to accomplish adaptive or perfective mainte-
nance, or is seeking reusable parts of a program.

Future work will be devoted to extend the recovery process to other pattern families, integrating dynamic analysis and model-checking in the present framework.
Bibliography


   http://hillside.net/patterns/ patterns.html.


CHAPTER V

Source Code Directory Structure Reorganization

5.1 Introduction

As described in Chapter I, software systems are subject to repeated maintenance interventions: new source files are added, some files are removed, changed, or moved. These interventions, especially emergency interventions, tend to deteriorate the original source code organization. When performing a new change, it will be difficult to locate the files related to a particular subsystem and, even worse, it could be difficult to rebuild the system since makefiles are not up-to-date.

This chapter describes a method to restructure the architectural source code files organization of a large software system. The method is based on the analysis of the lattice built using CA (Concept Analysis) from object module dependencies.

It represents a different point of view with respect to library reorganization described in Chapter VI. In fact, while the library identification/reorganization aims to cluster together files used by a common set of applications, here the aim is to exploit dependencies to reconstruct makefiles, to separate subsystem files and to highlight those files that are common to more subsystems. It is worth noting that the two approaches are not alternative, but complementary: as it will be shown, once library files have been identified, they will be put in a separate directory hierarchy.

The method was applied to public domain and freely available software systems developed with C programming language. A legacy system situation was mimicked by flattening out, when present, the directory organization (all files were placed in a
common directory). In other words, the worst-case situation, where all information was lost, was artificially obtained. By qualitatively comparing the new directory organization with the original one, a measure of the method performance was achieved. In the experimentation, since C does not have a construct to identify modules, it was assumed a correspondence between C source files and modules.

Results obtained are encouraging: the method was always able to identify, for each executable (i.e., main), all the other required modules to successfully compile a working system. Finally, the directory organization was judged very effective even if different from the original one.

The remainder of this chapter is organized as follows. After a review of the related work on CA application to software reengineering, a primer of CA is presented. Then, Section 5.4 introduces the proposed method and describes the tools used to perform the analysis. Section 5.5 presents the case study, followed by a discussion of the results obtained. The final section summarizes lesson learned.

5.2 Related Work

An overview of CA applied to software reengineering problems was shown by G. Snelting in his seminal work [1], where he used CA in several remodularization problems such as exploring configuration spaces (see also [2]), transforming class hierarchies, and remodularizing Cobol systems. In [3] Kuipers and Moonen combined CA and type inference in a semi-automatic approach to find objects in Cobol legacy code. As in [2, 3] the proposed approach is semi-automatic: the maintainer is left in charge to choose the proper remodularization based on his/her knowledge.

The need of a human intervention may be brought back to the size of the concept lattice, the number of concepts, links (in other words the number of discovered relations). The problem of redundant information returned by a concept lattice was illustrated by Anquetil [4] where he proposed methods to prune the lattice. Another approach for extracting significant information from a concept lattice was proposed by Siff and Reps [5], where they exhibit an algorithm to recover partitions from a concept lattice.
Commonalities can be found with [6] and [7]. In [6] a method for decomposing complex software systems into independent subsystems was proposed by Anquetil and Lethbridge. Source files were clustered based on file names and their decomposition. Merlo et al., [7] exploited comments, as well as variable and function names to clustered files. This work shares with the authors of [6, 7] the final goal: performing system remodularization during program understanding activities or when performing maintenance interventions.

Canfora et al. [8] used CA to identify objects into procedural code. Eisenbarth et al. [9, 10, 11, 12] applied CA on information obtained from both static and dynamic analysis to understand features present into source code. Finally, CA was applied to remodularize C system based on memory usage [13, 14] and to identify design patterns on C++ systems [15].

5.3 Background Notions: a Primer on Concept Analysis

CA is a mathematical tool that allows to identify groups of objects having common attributes. First CA study dates back to 1940, when G. Birkoff proved the possibility of construction of a lattice starting from binary relations between a set of objects and their attributes [16]. Since 1982 [17] there has been a flourishing of activities in several areas of software engineering: for example, CA has been used for remodularization of legacy code [18], for identifying objects [5] and software configurations [2].

CA can be thought of as the process of searching “rectangles” in a boolean table representing a relation between objects and attributes. Thus, a concept is a maximal rectangle in a table where columns and rows permutations are allowed. More precisely [19], CA starts with a context, a triple, $C = (O, A, R)$, where $O$ is a finite set of objects, $A$ is a finite set of attributes, and $P \subseteq O \times A$ is a relation between $O$ and $A$. If the pair $(o, a) \in P$, it can be said that object $o$ has attribute $a$. Given a set of objects $X \subseteq O$,

$$\sigma(X) := \{ a \in A \mid \forall o \in X : (o, a) \in P \}$$
is the set of common attributes while, given $Y \subseteq A$,

$$\tau(Y) := \{ o \in O \mid \forall a \in Y : (o, a) \in P \}$$

is the set of common objects.

These two mappings ($\sigma : 2^O \to 2^A \text{ and } \tau : 2^A \to 2^O$) form a Galois connection. Note that the mappings are antimonotone:

$$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1)$$

$$Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

and extensive:

$$X \subseteq \tau(\sigma(X)) \text{ and } Y \subseteq \sigma(\tau(Y)).$$

A concept is a pair of sets $(X, Y)$ where $X \subseteq O$ is called the extent, $Y \subseteq A$ is called the intent, and $Y = \sigma(X)$, $X = \tau(Y)$. That is, a concept is a maximal collection of objects sharing common attributes. The set of all concepts is denoted by $B(O, A, R)$. Furthermore, a concept $(X_1, Y_1)$ is a subconcept of another concept $(X_2, Y_2)$ if $X_1 \subseteq X_2$. This imposes a partial order relation on $B(O, A, R)$, and it can be written that $(X_1, Y_1) \preceq (X_2, Y_2)$. The partial order $\preceq$, can be used to build a lattice called concept lattice, where each node represents a concept. The concept lattice introduces a hierarchical clustering of objects and attributes, where suprema factor out common attributes, while infima factor out common objects. More details can be found in [19, 5].

## 5.4 The Method

As shown in Figure 5.3, the method is based on three sequential steps:

1. **Executables Identification and Module Dependencies Detection**: the main module of each system executable is detected; modules and external libraries required to produce each executable are identified;

2. **Library identification**: as said, this step is detailed in Chapter VI; and
3. **Directory Organization**: the source code files (e.g., .c and .h) are organized into a new directory hierarchy.

The following subsections details each step using the system described by Table 5.1 as a running example. The example is based on a hypothetical system having four different executables: **exe1, exe2, exe3** and **exe4**. The files required to compile each executable are shown in the second column of the table. Table 5.1 could be thought of as a compact representation of a CA context: objects are contained in the first column (the executables), attributes are the elements of the second column (objects required).

<table>
<thead>
<tr>
<th>Executable</th>
<th>Objects required</th>
</tr>
</thead>
<tbody>
<tr>
<td>exe1</td>
<td>f1, f14, f5, f1, f12, f13</td>
</tr>
<tr>
<td>exe2</td>
<td>f2, f23, f6, f1, f12, f13</td>
</tr>
<tr>
<td>exe3</td>
<td>f3, f23, f7, f11</td>
</tr>
<tr>
<td>exe4</td>
<td>f4, f8, f14, f11</td>
</tr>
</tbody>
</table>

Table 5.1: Objects needed to build each executable (including library files).
5.4.1 Executables Identification and Modules Dependencies Detection

This step, given a platform configuration, produces a makefile that generates all the executables constituting the analyzed software system. In particular, it works as follows: *main* modules of the executables composing the system are detected. For languages such as C and C++ *main* module is the object file containing the definition of the *main* symbol. Then, the object file of each executable main module is analyzed, and the list of unresolved symbols is retrieved. These symbols are recursively resolved, searching the definitions in the other object files: modules resolving symbols are tagged as required to build the executable. The analysis of the existing makefiles could be required whenever different declarations with the same name appear in more than one module.

Finally, (in addition to what described in Section 6.4.1) unresolved symbols are searched into libraries (both system libraries and third-part libraries) in order to determine which libraries must be linked to the executable. A similar process, except for the last step, will be adopted in Chapter VI to produce the dependency graph used for library remodularization.

<table>
<thead>
<tr>
<th>Executable</th>
<th>Source files required</th>
</tr>
</thead>
<tbody>
<tr>
<td>exe1</td>
<td>f1, f14, f5</td>
</tr>
<tr>
<td>exe2</td>
<td>f2, f23, f6</td>
</tr>
<tr>
<td>exe3</td>
<td>f3, f23, f7</td>
</tr>
<tr>
<td>exe4</td>
<td>f4, f8, f14</td>
</tr>
</tbody>
</table>

Table 5.2: Objects needed to build each executable (without library files).

5.4.2 Directory Organization

The directories containing the system source code files (i.e., .c and .h files) are grouped exploiting the makefile produced by the first step and taking into account the .o files grouped into libraries.

Firstly, the source code files related to the objects previously clustered into libraries are organized. In particular, a top level directory named *lib* is created: it contains a subdirectory, for each identified library, storing all the modules required
to generate the library itself. In the running example, two subdirectories are created: lib1 contains the source code related to f11, while lib2 contains the source code related to both f12 and f13 (see Figure 5.2).

<table>
<thead>
<tr>
<th>Id</th>
<th>Objects</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>{ex1}</td>
<td>{f11, f13, f14, f16, f17, f18}</td>
</tr>
<tr>
<td>c1</td>
<td>{ex2}</td>
<td>{f13, f16}</td>
</tr>
<tr>
<td>c2</td>
<td>{ex3}</td>
<td>{f13, f17, f18}</td>
</tr>
<tr>
<td>c3</td>
<td>{ex4}</td>
<td>{f14, f18, f19}</td>
</tr>
<tr>
<td>c4</td>
<td>{ex5, ex6, ex7, ex8}</td>
<td>{f16, f17}</td>
</tr>
<tr>
<td>c5</td>
<td>{ex9, ex10}</td>
<td>{f18}</td>
</tr>
<tr>
<td>c6</td>
<td>{ex11}</td>
<td>{f19}</td>
</tr>
<tr>
<td>c7</td>
<td>{ex12}</td>
<td>{f20}</td>
</tr>
</tbody>
</table>

Table 5.3: Sample example: concepts.

The directory organization of the remaining modules (i.e., the .c and .h files not required to produce the .o files clustered into libraries) follows an executables perspective. As above, the set of objects corresponds to the set of executables, while the set of attributes is obtained by collecting the .c and .h files. The binary relation is represented by the is-required-to-compile relation (information recovered by step 1 and 2).

Table 5.2 contains objects (i.e., .c and .h files) not previously clustered into libraries. Table 5.3 reports the concepts retrieved by the CA algorithm for the running example of 5.2.

The procedure used to derive the directory organization takes as input the set of concepts evaluated by the CA algorithm. The procedure firstly eliminates all the concepts having empty sets of objects and/or attributes (Table 5.3 concepts c0 and c5). Then, all the attributes that are shared by several concepts are identified (e.g., the attribute f14 is shared by c1 and c6). Common attributes are removed from all the concepts having a set of objects with cardinality equal to one. Giving raise to what called “pruned concepts”. Table 5.4 reports the pruned concepts corresponding to Table 5.3.

A hierarchy with two levels of directories is created. In particular, a directory is created for each pruned concept: the directory content is defined by the attribute set of the pruned concept (i.e., the set of .c files grouped by the pruned concept itself),
Table 5.4: Sample example: concepts after pruning common attributes.

while its hierarchy level depends on the cardinality of the object set of the pruned concept (i.e., the set of executables grouped by the pruned concept).

The directories belonging to the first level are obtained by the pruned concepts having a set of objects with cardinality exactly equal to one. In fact, such concepts group for each executable the maximum number of .c file required exclusively to compile that executable. It is worth noting that the procedure always creates a number of first level directories equal to the number of executables. In fact, there will always be a pruned concept having a set of objects with cardinality equal to one for each system executable, due to the main module.

In the running example, four first-level directories (see Figure 5.2) are created starting from Table 5.4: dir-exe1, dir-exe2, dir-exe3 and dir-exe4.

The directories belonging to the second level are created taking into account the pruned concepts having a set of objects with cardinality greater than one. Such concepts capture an important class of information: they actually group the maximum set of executables requiring for the compilation the same set of .c files. In other words, such concepts highlight cross-directory/cross-sub-system dependencies. For example, the concept c6 highlights that the compilation of both exe1 and exe4 requires the file f14. In this case a physical directory, dir-exe1_exe4, is created into dir-exe1 and it is symbolically linked to dir-exe4. The directory dir-exe1_exe2 will contain the file f14 (see Figure 5.2).

Finally .h files are organized: in each first level directory previously created a subdirectory, named include is added. The include subdirectory will hold all the
.h files directly included by the source files contained in the first level directory.

It is worth noting that whenever a .c or a .h file is required by several directories it will physically be contained in one only directory, then a symbolic link will be created to all the other directories requiring it.

![Diagram of directory organization](image)

Figure 5.2: Directory organization (dotted lines are intended as symbolic links).

### 5.4.3 Tool Support

A Perl script was developed to support the reconstruction of the system makefile; it uses the nm Unix utility to identify both the defined and undefined symbols used by each .o file. The script takes as input the .o file containing the main function and it begins to extract unresolved symbols; it continues recursively, analyzing all the other .o files in the same directory, resolving some symbols and, at the same time, finding new unresolved symbols. Finally, the script searches for libraries into directories specified in a configuration file to resolve remaining (unresolved) symbols. The output produced is represented by the list of .o files and libraries that have to be linked to the .o file containing the function main to obtain an executable application.

CA lattice was built using the tool concepts [20]. The tool takes as input an objects/attributes matrix and produces a textual lattice representation. Then, the lattice can be displayed and navigated using a dot [21] file viewer, such as At&T Dotty [22].
5.5 Case Studies

To assess the feasibility and the effectiveness of the approach, several versions of two software systems, mSQL and Samba, were considered.

5.5.1 Case Study Description

mSQL is a is a relational database system; more details can be found in Section 2.5.2.

Samba (http://www.samba.org) is a freely available file server that runs on Unix and other operating systems (usually to share resources between Unix-based systems and Microsoft-based systems). The code has been written to be as portable as possible. It has been “ported” to many unices (Linux, SunOS, Solaris, SVR4, Ultrix, etc.).

Samba consists of two key programs, smbd and nmbd, plus a bunch of other utilities. They implement four basic services: file sharing & print services, authentication and authorization, name resolution and service announcement (browsing). Moreover, Samba comes with a variety of utilities. The most commonly used utilities are: 
smbclient, a simple SMB (Service Message Block - a protocol for sharing general communications abstractions such as files, printers, etc.) client; nmblookup a NetBIOS name service client, and swat which is the Samba Web Administration Tool, that allows the configuration of Samba remotely, using a web browser.

15 releases of Samba (from 1.9.00 to 1.9.08 and from 2.0.0 to 2.0.5) and 11 releases of mSQL (from 1.06 to 1.08 and from 2.0.1 to 2.0.8) were analyzed. The results obtained did not significantly vary within each family (two releases belong to the same family if the first two version numbers are the same, i.e., 1.9.00 and 1.9.08 are considered to be members of the same family), for this reason a single release, extracted from each family, is discussed in this section: it can be thought of as representative of the entire family.
5.5.2 Case Study Results

Tables 5.6 and 5.5 lists, for each representative system, the results obtained by applying step 1 of the reorganization process. In particular, for each executable, the following identified information is shown:

1. The number of defined symbols;

2. The number of modules required for a successful compilation and linking, in parenthesis is reported such a number as extracted by the original system Makefile; and

3. The number of external libraries required for a successful compilation and linking. In parenthesis it is reported the number as extracted by the original Makefile.

It is worth noting that all the modules and the libraries necessary for a successful compilation and linking were always retrieved.

Tables 5.6 and 5.5 highlight that both the analyzed systems had, during the time, dramatic changes. The changes implemented on both the systems added new functionalities, improved existing features and reorganized systems architecture. However, such a reorganization for mSQL required a decreasing of the modules that each executable has to link, while for Samba the reorganization implied an increasing of the linked modules.

As it can be seen in Table 5.5, the number of objects used to build Samba 2.0.5 applications is usually smaller than the retrieved from the Makefile. This happens because whenever a symbol defined in a library is needed, developers linked the entire library (composed of several object files), even if a unnecessary larger executable was obtained. On the contrary, the proposed method takes into account only objects effectively needed to resolve symbols.

The original directory organization¹ implemented by both Samba and mSQL developers follows the de-facto Unix standard directory organization: separate di-

¹Samba 1.9.08 had a completely flat structure: all the files were contained in one single directory
<table>
<thead>
<tr>
<th>Samba 1.9.08</th>
<th>Samba 2.0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symbols defined</strong></td>
<td><strong>Objects to link</strong></td>
</tr>
<tr>
<td>smbclient</td>
<td>190</td>
</tr>
<tr>
<td>smb</td>
<td>384</td>
</tr>
<tr>
<td>nmbd</td>
<td>130</td>
</tr>
<tr>
<td>nmblookup</td>
<td>-</td>
</tr>
<tr>
<td>rpcclient</td>
<td>-</td>
</tr>
<tr>
<td>smbpasswd</td>
<td>-</td>
</tr>
<tr>
<td>smbstatus</td>
<td>156</td>
</tr>
<tr>
<td>smbprep</td>
<td>11</td>
</tr>
<tr>
<td>swat</td>
<td>-</td>
</tr>
<tr>
<td>testparam</td>
<td>119</td>
</tr>
<tr>
<td>testsrm</td>
<td>163</td>
</tr>
<tr>
<td>make_printerdef</td>
<td>-</td>
</tr>
<tr>
<td>make_smbcodepage</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.5: Samba module dependencies.

Directories are created for include files, applications source code, libraries and binary files. Such an organization is mainly thought for the end-user who is not in charge of the software application maintenance and evolution. On the other hand, the new directories organization aims to support maintenance and evolution activities.

The comparison between the new directory organization and the actual directory structure of the analyzed systems was encouraging. In particular, whenever the original developers grouped into distinct directories the source code files related to the system executables, the directory organization proposed matched the actual one. For example, the original Samba 2.0.5 developers placed the .c files related to four executables (nmbd, swat, smb and rpcclient) into four distinct directories. In this case there is a perfect correspondence between the .c files contained in such actual directories and the .c files contained in the 4 directories created by the method and associated to those 4 executables. On the other hand mSQL developers, for both the analyzed system versions, concentrated almost 50% the source code related to the system executables in a single directory (src/m SQL). In particular, for mSQL 1.0.6. all the source code files related to the system executables were contained in src/m SQL. In mSQL 2.0.8 such a directory contains the source code files related to 9, on a total of 12, executables. In this case the retrieved directory organization differs from the actual one.
However, redefining the directory organization aims mainly to highlight cross-directory/cross-subsystem dependencies and, even when it differs from the original one, it is not less effective. In particular, highlighting dependencies may be helpful for software systems frequently evolved. Two principal benefits may derive from this organization:

1. When a system executable has to be modified the suggested directory organization allows the quick identification of the source code files of the executable; and

2. Grouping together source code files shared by different executables may help to avoid undesired side effects due to source code modifications. In other words, whenever a source code file is modified, the suggested directory organization highlights all the executables potentially impacted by such a modified file.

<table>
<thead>
<tr>
<th></th>
<th>mSQL 1.0.6</th>
<th></th>
<th></th>
<th>mSQL 2.0.8</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Defined</td>
<td>Objects</td>
<td>Libraries</td>
<td>Defined</td>
<td>Objects</td>
<td>Libraries</td>
</tr>
<tr>
<td></td>
<td>symbols</td>
<td>to link</td>
<td>to link</td>
<td>symbols</td>
<td>to link</td>
<td>to link</td>
</tr>
<tr>
<td>mSQL</td>
<td>61</td>
<td>2 (2)</td>
<td>1 (1)</td>
<td>65</td>
<td>4 (4)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>mSQLd</td>
<td>110</td>
<td>9 (9)</td>
<td>2 (2)</td>
<td>140</td>
<td>26 (23)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>mSQLadmin</td>
<td>88</td>
<td>9 (4)</td>
<td>2 (2)</td>
<td>62</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>relshow</td>
<td>87</td>
<td>9 (3)</td>
<td>2 (2)</td>
<td>57</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>insert_test</td>
<td>45</td>
<td>2 (2)</td>
<td>1 (1)</td>
<td>52</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>select_test</td>
<td>45</td>
<td>2 (2)</td>
<td>1 (1)</td>
<td>52</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>mSQLdump</td>
<td>89</td>
<td>9 (4)</td>
<td>2 (2)</td>
<td>60</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>mSQLimport</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>62</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>mSQLexport</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>55</td>
<td>3 (3)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>lite</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>164</td>
<td>41 (19)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>w3-mysql</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>147</td>
<td>41 (21)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>w3-auth</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>59</td>
<td>5 (6)</td>
<td>1 (1)</td>
</tr>
</tbody>
</table>

Table 5.6: mSQL module dependencies.

5.6 Conclusions

The proposed method recovers an architectural source code files organization analyzing compiling, linking and inclusion dependencies and using CA. The method has
been applied to software systems written with the C programming language. The results obtained are encouraging: it was always possible to reconstruct the Makefiles and to generate the executables of the analyzed systems. The new directory organization highlights dependencies, thus it may help to maintain and evolve systems frequently modified.
Bibliography


CHAPTER VI

Library Identification and Miniaturization

6.1 Introduction

Porting existing software applications on hand-held devices, such as PDA (Personal Digital Assistants) or wireless devices (e.g., multimedia cellphones) is one of the new software trends and industry hip. Although wireless technology has reached a turning point, where complex applications can be ported on small devices, the resources available are often limited, if compared to the platforms for which the applications were conceived.

At the same time, one of the undesired effects of software evolution is the proliferation of unused components, or components unlikely to be used by a given subset of the applications. As a consequence, the size of binaries and libraries tends to grow.

To go wireless applications face the miniaturization challenge: the extra fat must be eliminated to ensure space and bandwidth optimization. Clearly, several actions may be taken. First and foremost, dead code should be removed and clones should be factored out. Furthermore, some form of restructuring, at library and at object file level, may be required. The latest intervention must deal with dependencies among software artifacts.

In summary, this chapter identifies and describes actions devoted to software system miniaturization, composed of the different above mentioned activities: removing unused objects and clones, minimizing dependencies, moving to smaller libraries, and identifying new libraries. The main advantage of the proposed approach is its
language independence: all the activities (except clone detection) rely on information extracted from object files; furthermore the clone detection algorithm adopted is not tied to specific programming languages (provided that a set of metrics can be extracted from the source code).

For any given software system, dependencies among executables and object files may be represented via a dependency graph, a graph where nodes represent resources and edges resource dependencies. Each library, in turn, may be thought of as a subgraph in the overall object file dependency graph. Therefore, software miniaturization can be modeled as a graph partitioning problem. Unfortunately, it is well known that graph partitioning is an NP-hard problem [1] and thus often heuristics have been adopted to find a sub-optimal solution. For example, one may be interested to first examine graph partitions minimizing cross edges between sub-graphs corresponding to libraries. More formally, a cost function describing the restructuring problem has to be defined, and heuristics driving the solution search process must be identified and applied.

The proposed approach stems from the observation that previously proposed approaches to software miniaturization were not completely satisfactory. For example, it is not obvious if pruning clones may be beneficial to reduce the memory requirements of executables. This, in fact, could increase coupling between libraries. Moreover, a library miniaturization approach based solely on traditional clustering techniques may be unable to find solutions easily identified by GA. Conversely, GA requires a starting population; choosing a random solution may not be very effective, or it may lead to a local sub-optimal solution. To overcome the aforementioned limitations, this chapter proposes a novel approach where an initial sub-optimal solution for new libraries (i.e., a set of graph partitions) is determined via traditional clustering approaches, followed by a GA search aimed at reducing the inter-library dependencies. GA is applied to a newly defined problem encoding, where genetic mutation may lead sometimes to generate clones, clones that do indeed reduce the overall amount of resources required by the executables, in that they remove inter-library dependencies. Performance improvement was also achieved by means of a
hybrid approach, obtained combining a GA with hill climbing techniques.

The framework exploits a multi-objective fitness function, trying to minimize, at the same time, both the number of inter-library dependencies and the average number of objects linked by each application. Finally, the fitness function accounts for the expert suggestions, leading to a semi-automatic approach composed of multiple iterations of the miniaturization process, interleaved by expert feedback.

The framework also encompasses the possibility of library miniaturization using dynamic information [2]. The approach stems from the fact that static information gives a limited view of the complex dependencies and relationships existing between different components of a software system. In fact, statically computed dependencies may not reflect the real user profile, leading to unnecessary large executables. Meanwhile, if, for example, a static calling dependency between two functions exists, then there may exist rare environment and parameter configurations forcing a function to call the other. On the other hand, dynamic information reflects the in field application use, accounting for it when miniaturization will lead to a more realistic, user profile-oriented, result.

It is therefore suggested, where needed, to linearly combine static and dynamic information (i.e., dependencies) with a weight, expressing the believe about the frequency of rare events. In other words, static information ensures to keep into account dependencies not dynamically exploited during the execution of the instrumented code.

The miniaturization process has been applied to a large (517 applications and 43 libraries, for a total of over 1 million LOCs) Open Source software system, GIS (Geographical Information System) named GRASS (Geographic Resources Analysis Support System) (http://grass.itc.it). GRASS is a raster/vector GIS combined with integrated image processing and data visualization subsystems [3]. For the dynamic miniaturization, results are available for public domain software applications such as Samba and MySQL. Examples of library identification with CA are shown for Samba and mSQL.

The chapter is organized as follows. First a short review on related work (Sec-
tion 6.2) and on the main notions of clustering and GA (Section 6.3), will be presented. Then, the phases of the library miniaturization process will be described in Section 6.4. Sections 6.6, 6.7 and 6.8 report results obtained applying the miniaturization process, library identification using CA and dynamic miniaturization respectively. The last section summarizes conclusions and outlines future developments.

6.2 Related Work

Many works are reported in literature concerning with software system modules clustering and/or restructuring, identifying objects, and recovering or building libraries. Most of these works applied clustering or CA (an overview of CA software reengineering applications is reported in Section 5.2).

A comparison between clustering and CA was presented in [4]. The library miniaturization approach proposed in this chapter shares with [4] the idea of applying an agglomerative-nesting clustering to a Boolean usage matrix, although in [4] the matrix indicated the uses of variables by programs. A survey of clustering techniques applied to software engineering was presented by Tzerpos and Holt in [5]. The same authors presented in [6] a metric to evaluate the similarity of different decompositions of software systems, in [7] a clustering algorithm oriented to program comprehension, and they discussed in [8] the problem of stability of software clustering algorithms. Another overview of cluster analysis applied to software systems has been presented in [9].

Applications of clustering to reengineering were suggested in [10] and [11]. In [10] a method for decomposing complex software systems into independent subsystems was proposed by Anquetil and Lethbridge. Source files were clustered according to file names and their decomposition. Merlo et al. [11] exploited comments, as well as variable and function names to cluster files. An approach relying on inter-module and intra-module dependency graphs to remodularize software systems was presented in [12]. As in [12] the library miniaturization approach presented in this chapter relies on the idea of analyzing dependency graphs, finding a tradeoff between having highly cohesive libraries and a low inter-connectivity.
GAs have been recently applied in different fields of computer science and software engineering. An approach for partitioning a graph using GA was discussed in [13]. Similar approaches were also shown in [14, 15, 16]. Maini et al. [17] discussed a method to introduce the problem knowledge in a non-uniform crossover operator, and presented some examples of its application. GA was used by Doval et al. [18] to identify clusters on software systems. Finally, Harman et al. [19] reported experiments of modularization/remodularization, comparing GA with hill climbing techniques, and introducing a representation and a crossover operator tied to the remodularization problem. Their case studies revealed that hill climbing outperformed GA.

Preliminary results from the GRASS reorganization and library miniaturization were proposed in [20], where several activities were carried out in order to miniaturize GRASS libraries. In particular, unused symbols were identified and pruned, clones were removed, and a preliminary work aimed at splitting the largest libraries was performed. The library miniaturization process was then improved in [21], determining the optimal number of clusters applying the Silhouette statistics and minimizing the number of inter-cluster dependencies using GA.

Recently, several works have considered a new perspective taking into consideration both static and dynamic information extracted from software artifacts [22, 23, 24] to locate concepts, to define coupling, or to support maintenance and program comprehension activities. In [25] T. Systä presented a tool for extracting scenarios from traces obtained from debugging Java bytecode. Similarly, in [26] authors used dynamic information to recovery collaboration diagrams and roles. An automatic technique for extracting operation sequences was shown in [27], while [28] described an approach, based on trace analysis, to discover thread interactions. The idea of using dynamic information for clustering was proposed in [29], where authors proposed a tool, Gadget, for extracting the dynamic structure of Java programs. With respect to [29], the dynamic miniaturization approach proposed in this chapter aims to to combine static and dynamic information, in that often not all dependencies and uses are exploited when executing instrumented programs.
6.3 Background Notions

The proposed approach for library miniaturization requires to integrate clustering and GA techniques in a semi-automatic, human-driven process. Clustering deals with the grouping of large amounts of things (entities) in groups (clusters) of closely related entities [30, 31, 32]. Clustering is used in different areas, such as business analysis, economics, astronomy, information retrieval, image processing, pattern recognition, biology, and others. GAs come from an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems. GAs are applied to different domains such as machine and robot learning, economics, operations research, ecology, studies of evolution, learning and social systems [33].

Remodularization approaches alternative to clustering rely on CA. Although, as said, the proposed approach relies on clustering and GA rather than CA, this chapter will also present an example of library identification with CA (see Section 6.7). This example constitutes a good clarification point in that, as it will be clearer later, the CA based approach provides an immediate visualization of object dependencies.

The following subsection reports essential background notions on clustering and GA. A primer on CA was presented in Section 5.3.

6.3.1 Clustering

In this work, the agglomerative-nesting (agnes) algorithm [34] was applied to build the initial set of candidate libraries. Agnes is an agglomerative, hierarchical clustering algorithm: it builds a hierarchy of clusters in such way that each level contains the same clusters as the first lower level, except for two clusters, which are joined to form a single cluster. In particular, agglomerative algorithms start building the dendrogram from the bottom of the hierarchy (where each one of the \(N\) entities represents a cluster), until at the \(N - 1\) level all entities are grouped in a single cluster.

The key point of hierarchical clustering is determining the cut-point (see Figure 6.3.1), i.e., the level to be considered in order to determine the actual clusters.
As it will be shown in Section 6.3.1.1, in this work such operation was supported by the Silhouette statistics.

![Hierarchical clustering dendrogram and cut-point.](image)

**Figure 6.1:** Hierarchical clustering dendrogram and cut-point.

### 6.3.1.1 Determining the Optimal Number of Clusters

To determine the actual or optimal number of clusters, traditionally, people rely on the plot of an error measure representing the within cluster dispersion. The error measure decreases as the number of cluster $k$ increases, while for some values of $k$ the curve flattens. Traditionally, it is assumed that the error curve *elbow* indicates the appropriate number of clusters [35]. To overcome the limitation of such a heuristic approach, several methods have been proposed (see [35] for a comprehensive summary).

Kaufman and Russseeuw [34] proposed the *Silhouette* statistics for estimating and assessing the optimal number of clusters. For the observation $i$, let $a(i)$ be the average distance to the other points in its cluster, and $b(i)$ the average distance to points in the nearest cluster (but its own), then the Silhouette statistics is defined as:

$$ s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{6.1} $$

Kaufman and Rouseeuw suggested choosing the optimal number of clusters as the value maximizing the average $s(i)$ over the dataset. Notice that the Silhouette statistics, as most of the methods described in [35], has the disadvantage that it
is undefined for one cluster, and thus it offers no indication of whether the current dataset already represents a good cluster. Since our purpose is to split the original libraries into smaller ones, this does not constitute a problem.

### 6.3.2 Genetic Algorithms

GAs revealed their effectiveness in finding approximate solutions for problems where:

- The search space is large or complex;
- No mathematical analysis is available;
- Traditional search methods did not work; and, above all
- The problem is NP-complete or NP-hard [1, 13].

Roughly speaking, a GA may be defined as an iterative procedure that searches for the best solution of a given problem among a constant-size population, represented by a finite string of symbols, the *genome*. The search is made starting from an initial population of individuals, often randomly generated. At each evolutionary step, individuals are evaluated using a *fitness function*. High-fitness individuals will have the highest probability to reproduce themselves.

The evolution (i.e., the generation of a new population) is made by means of two kinds of operator: the *crossover operator* and the *mutation operator*. The crossover operator takes two individuals (the *parents*) of the old generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*). The mutation operator has been introduced to prevent convergence to local optima, in that it randomly modifies an individual’s genome (e.g., flipping some of its bits if the genome is represented by a bit string). Crossover and mutation are respectively performed on each individual of the population with probability *pcross* and *pmut* respectively, where *pmut* \(< *pcross*.

The GA does not guarantee to converge: the termination condition is often specified as a maximal number of generations, or as a given value of the fitness function. The GA behavior can be represented by the pseudo-code shown in Figure 6.3.2.
Initialize population $P[0]$; 
g=0; // Generation counter 
while(g < max_number_of_generations) 

  //Apply the fitness function to the 
  //current population 
  Evaluate $P[g]$; 

  //Advance to the next generation 
  g=g+1; 

  //Make a list of pairs of individuals 
  //likely to mate (best fitness) 
  Select $P[g]$ from $P[g-1]$; 

  //Crossover with probability 
  //pcross on each pair 
  Crossover $P[g]$; 

  //Mutation With probability 
  //pmut on each individual 
  Mutate $P[g]$; 

end while 

Figure 6.2: Pseudo-code of a GA. 

6.3.2.1 Hill Climbing and GA Hybrid Approaches

As suggested in literature [33], hybrid GAs may reveal advantageous when there is the need for optimization techniques tied to specific problem structure. The *in*-
*large* perspective of GA may be combined with the precision of local search. GAs are able to explore a large search space, but often they reach a solution that is not accurate, or they converge to an accurate solution very slowly. On the other hand, local optimization techniques (such as hill climbing) quickly converge to a local optimum, but they are not very effective to search in large solution spaces (in that they suffer of problems such as local maximum or plateaus).

There are different ways to hybridize a GA with hill climbing techniques. The first approach attempts to optimize, using hill climbing techniques, the best individuals of the last generation. The second approach uses hill climbing to optimize the best individuals of each generation. Applying hill climbing on each generation could be expensive, however this technique “inserts” in each generation a *high quality individual*, obtained from the optimization, reducing therefore the number of generations requested to ensure GA convergence.
6.4 The Library Reorganization Process

As highlighted in the introduction, the proposed library reorganization process consists of several steps:

1. First and foremost, software system applications, libraries and dependencies among them are identified;

2. Unused functions/objects are identified and, if needed, removed from the system or, possibly, stored in an appropriate repository;

3. Duplicated (cloned) objects are identified and, whenever possible, factored out;

4. Circular dependencies among libraries are removed, or, at least, minimized. In fact, these dependencies cause a library to be linked each time another one (circularly linked to it) is needed;

5. Large libraries are split into smaller ones and, if possible, transformed into dynamic libraries; and

6. Objects used by multiple applications (not yet organized into libraries) are grouped into new libraries.

The activities of the miniaturization process, as well as the graph representation they rely on, are detailed in the following subsections.

6.4.1 Software System Graph Representation

Central to the miniaturization process is the software system representation; most of the computation activities rely on a graph highlighting dependencies between object modules \( O \equiv \{o_1, o_2, \ldots, o_p\} \). The software system may be, in fact, represented by the SG (System Graph) defined as:

\[
SG \equiv \{O, D\} \quad (6.2)
\]
where $D \subseteq O \times O$ the set of oriented edges $d_{i,j}$ representing dependencies between objects. An example of SG is depicted in Figure 6.3. Nodes of the SG may be classified in two categories:

1. Source\(^1\) nodes containing the main symbol, drawn in Figure 6.3 as squares, representing the set of the $m$ software system applications $A \equiv \{a_1, a_2, \ldots, a_m\}$; and

2. Other nodes, indicated as circles, representing objects needed by applications. Some of these objects may be contained into libraries (depicted as rounded boxes). The set of $n$ system libraries is indicated as $L \equiv \{l_1, l_2, \ldots, l_n\}$.

![Figure 6.3: Example of System Graph.](image)

It is possible to extract from the SG other graphs useful for clustering purposes. The first graph, the *Use Graph*, highlights uses of objects by applications or by libraries. The *use* relationship is defined as:

\(^1\)It is worth noting that applications are not the only source nodes. In fact, as it will detailed later, also unused objects have no incoming edges, even if they can be distinguished from the applications since the latter also define a main symbol.
\[ a_x \text{ uses } o_y \iff \exists \text{ path } \{a_x, \ldots, o_y\} \in SG \quad (6.3) \]
\[ l_x \text{ uses } o_y \iff \exists o_j \in l_x \mid \exists \text{ path } \{o_j, \ldots, o_y\} \in SG \quad (6.4) \]

The *Use Graph* can be represented by a MU (Matrix of Uses):

\[
mu_{x,y} = \begin{cases} 
  1 & \text{if } x \leq m \text{ object } o_y \text{ is used by application } a_x \\
  & \text{by application } a_x \\
  0 & \text{otherwise} \\
  1 & \text{if } x > m \text{ object } o_y \text{ is used by library } l_{x-m} \text{ and } o_y \notin l_{x-m} \\
  & \text{by library } l_{x-m} \text{ and } o_y \notin l_{x-m} \\
  0 & \text{otherwise} 
\end{cases}
\]

The second graph, the *Dependency Graph*, is used to represent dependencies existing between two or more libraries, or between objects contained in a library to be split (the clustering algorithm should avoid inter-cluster dependencies). A *Dependency Graph* is simply a subgraph of SG composed of objects among which we want to analyze dependencies, and edges connecting them. Again, the graph may be represented as an adjacency matrix MD (Matrix of Dependencies), such that:

\[
md_{x,y} = \begin{cases} 
  1 & o_x \text{ depends from } o_y \\
  0 & \text{otherwise} 
\end{cases}
\]

### 6.4.2 Graph Construction

Prior to recover dependencies between applications and libraries, and between libraries themselves, the applications (i.e., executables) composing the software system must be identified. This work relies on an approach similar to the one proposed in Chapter V.
Once applications and existing libraries are identified (for the latter the identification process is trivial, in that it consisted in simply searching for .a files), the SG is built. Given the use relationship between an object module requiring a symbol and a module defining it, the graph is built via the transitive closure of the use relationship, starting from the main object of each application and from each library. Finally, the use graph and the dependency graph (and therefore the MU and MDs) are extracted from the SG.

6.4.3 Handling Unused Objects

Symbols defined in libraries, but not used by applications nor by other libraries, are likely to constitute useless resources, thus they should be identified. Their presence is often due to utility functions inserted in libraries (not used by the current set of applications), or to features not yet fully implemented.

The objects defining these symbols should be removed from the libraries, providing that objects do not also export used symbols (in that case the object should be left into library, or the corresponding source file restructured). One possible solution is to create, from each library, two new libraries, one of which containing all the unused symbols.

In any case, even if developers decide to leave these objects in their position, maybe because they can potentially be used by future applications (or new versions of old applications), their presence and their impact on the software system size should be highlighted.

6.4.4 Removal of Circular Dependencies among Libraries

The dependency graph captures dependencies between the different libraries and allows to identify strongly connected components. In particular, circular dependencies between libraries cause a library to be linked each time the other one is needed. Once these dependencies are identified, four choices could be taken to remove them:

1. Move the object causing the circular dependency from a library to another.

   This is only feasible if the object does not need resources located in its original
library, nor it is needed by that library. For example, in Figure 6.4-a, object o1 can be moved from library L1 to library L2;

2. **Duplicate the object**: similarly to the previous case, this is feasible if the object does not need resources located in the original library but, differently from the previous case, it is required in that library (therefore moving it outside worsen the situation). In Figure 6.4-b, object o1 should be duplicated in library L2 (it cannot be moved, in that o2 depends on it);

3. **Merge the two libraries**: this strategy should be avoided whenever possible; however, it could be the only available solution when the number of objects causing circular and, in general, inter-library dependencies is very high; or

4. **Make dynamic libraries**: instead of merging circularly dependent libraries, one may decide to make them dynamic. Circular dependency problem is not solved, but the average amount of resources needed is reduced (see details in Section 6.4.7.1).

![Diagram of dependencies among libraries](image_url)

Figure 6.4: Examples of dependencies among libraries.

When the *dependency graph* analysis does not allow to remove circular dependencies and, for performance reasons, options three and four cannot be adopted, a deeper analysis should be performed, identifying dependencies at function grain-level
instead of object grain-level. This should ease the removal of some critical dependencies. Let us consider the example in Figure 6.4-c: object o1 cannot be moved, nor duplicated in L2 (in that it depends from the object o2). However, splitting o1 into two chunks, leaving the function f2 in library L1, and moving f1 in L2, would solve the problem.

Finally, the existence of a complex dependency relationship between two libraries indicates the possibility (to be confirmed by developer’s feedback) of poor library design. In this case, library objects should be merged and then split again in new clusters, adopting the process detailed in Section 6.4.6.

6.4.5 Identification of Duplicates Symbols and Clones

Comparing the list of symbols defined in each library allows to detect the list of duplicated exported symbols. It is worth noting that homonym symbols in different libraries may refer to completely different functions, external variables or data structures. On the other hand two or more symbols, although having different names, may correspond to cloned functions.

Therefore, a clone analysis is necessary. In this context a metric-based clone detection process (see Section 2.3.2), aiming at detecting duplicated functions, is adopted. The results obtained may suggest different possible actions:

1. If a whole, duplicated, object module has been detected inside two or more libraries, then it should be left in only one of these;

2. If duplicated functions are identified inside different objects, refactoring could be performed moving them outside, applying considerations similar to the previous case; and

3. Finally, clone detection may reveal clones outside libraries, in that several applications may contain, in their objects, duplicated portions of code. In some cases, it should be useful to factor such duplicated code in a library.

Preliminary to the above described actions is an analysis of the impact in terms of dependencies (and, above all, circular dependencies) introduced. As explained
in Section 6.4.4 and as it will be shown in Section 6.4.6, sometimes an object is
duplicated to reduce dependencies. In general, it may be preferable to duplicate
few objects, rather than introducing a dependence that causes, for a subset of the
applications, the linking or the loading (if using DLLs) of one or more additional
libraries. It is worth noting that, if the process duplicates a conspicuous number
of objects into two or more libraries, these objects can be factored, as explained in
Section 6.4.7.1, into a new library from which the old libraries depend.

6.4.6 Library Miniaturization

The last, and most relevant point of the proposed process is devoted to split
existing, large libraries into smaller clusters, thus reducing the memory footprint
of applications. Basically, the idea has been proposed in [36]: objects used by a
common set of programs should be grouped together, trying to minimize the average
number of libraries required by each program.

In [36] the library identification was thought of as a CA problem, where the set
of objects corresponds to the set of executables, the set of attributes is obtained by
collecting the set of object files. The binary relation is represented by the is-required-
to-link relation.

<table>
<thead>
<tr>
<th>Executable</th>
<th>Objects required</th>
</tr>
</thead>
<tbody>
<tr>
<td>exe1</td>
<td>f1, f14, f5, f11, f12, f13</td>
</tr>
<tr>
<td>exe2</td>
<td>f2, f23, f6, f11, f12, f13</td>
</tr>
<tr>
<td>exe3</td>
<td>f3, f23, f7, f11</td>
</tr>
<tr>
<td>exe4</td>
<td>f4, f8, f14, f11</td>
</tr>
</tbody>
</table>

Table 6.1: Objects needed to build each executable (including library files).

The library identification process is based on the observation that object files
occurring in concepts close to the top (or listed in the top itself) are used by many
(all) executables. Therefore, concepts near to the top element of the concept lattice
can be considered good candidates for the creation of a library. For example, the
object f11 in Table 6.1, being used by all executables, is considered as a library;
furthermore, f12 and f13 are used by exe1 and exe2, so they also are clustered into
a different library. In some cases, clustering into a library of a large number of object files used by few executables may be convenient. This requires the analysis of the concepts near the bottom element of the concept lattice.

Although the lattice gives useful information (often good libraries are the sets of objects located on top nodes, or concepts retaining large percentages of objects), it becomes unmanageable when a large number of applications and libraries must be handled [37].

Moreover, while the construction of concept lattice is performed automatically from results of the previous step, lattice analysis requires a manual inspection to verify the usefulness of the candidate libraries. Once candidate files to build libraries are identified, each object file is assigned to a library according to the concept position in the lattice (i.e., depth first traversal). This avoids assigning a file to more than one library.

Instead of pruning information on concept lattice like [38, 39], a clustering analysis is performed, similarly to [10, 11, 12]. For each library $l_x$ to be split, a MU matrix is built from the subgraph of the use graph representing uses of $l_x$ objects from applications (and other libraries), and a MD matrix representing dependencies between all library objects.

Given this, the miniaturization works according to the process shown in Figure 6.5, consisting of four steps:

1. Determine optimal number of clusters;
2. Determine a sub-optimal solution by hierarchical clustering;
3. Determine the new candidate libraries using GA; and
4. Ask feedback to developers and, eventually, iterate through step 3.

6.4.6.1 Determining the Optimal Number of Clusters

As explained in Section 6.3.1.1, the optimal number of clusters is computed on each MU matrix inspecting the Silhouette statistics. Giving the curve of the average
Figure 6.5: Activity diagram of the library miniaturization process.

Silhouette values for different numbers $k$ of clusters, instead of considering the maximum (often too high for our miniaturization purpose), the optimal value is identified in correspondence of the \emph{elbow} of that curve [34]. The process also incorporates in the choice experts’ knowledge, and a tradeoff between excessive fragmentation and library size have been considered. Examples of Silhouette statistics are shown in Figure 6.10.

6.4.7 Determining the Sub-Optima Libraries by Hierarchical Clustering

Once known the number of clusters for each \emph{“old library”}, agglomerative-nesting clustering is performed on each MU matrix. This builds a \textit{dendrogram} and a vector of heights, that allow identifying $k$ clusters. These clusters constitute the new \textit{candidate}
libraries.

To assess the effectiveness of the miniaturization process, a measure of quality for the new libraries must be defined. Let \( k \) the number of clusters \( l_{x_1}, \ldots, l_{x_k} \) obtained from a library \( l_x \). Then, the \( PR_x \) (Partitioning Ratio) can be defined as:

\[
PR_x = 100 \sum_{i=1}^{m} \sum_{j=1}^{k} \frac{|l_{x_j}| * m u_{i,x_j}}{|l_x| * m u_{i,x}}
\]  

(6.5)

where \(|l_x|\) is the number of objects archived into library \( l_x \). The smaller is the PR, the most effective is the partitioning, in that the average number of objects linked (or loaded) by each application is smaller than using the whole old library.

### 6.4.7.1 Reducing Dependencies using Genetic Algorithms

The solution reached at the previous step presents two main drawbacks:

1. The number of dependencies between the new libraries could be high, forcing to load another library each time a symbol from that library is needed, and therefore wasting the advantage of having new smaller libraries; and

2. The new libraries may not be meaningful with respect to developer’s intentions: their feedback has to be incorporated in the miniaturization process.

Of course, an important step to perform is to convert static libraries to dynamic-loadable libraries, so that each (small) library is loaded at run-time only when needed, and then unloaded when it is no longer useful. In this case, even if there are dependencies among libraries, the average number of libraries in memory is considerably reduced with respect to the original system.

The removal of inter-library dependencies can be brought back to a graph partitioning problem that, as shown in [13], is NP-hard, and a GA was used to reach an approximate solution of the problem (i.e., minimizing the number of dependencies).

A GA requires the specification of:

1. The genome encoding;
2. The initial population;

3. The fitness function;

4. The crossover operator;

5. The mutation operator; and

6. All GA parameters, such as the crossover and mutation probability, the population size and the number of generations.

The encoding schema widely adopted in literature [13, 18] indicates each partition with an integer $p$ such that $0 \leq p \leq k - 1$ (where $k$ is the number of candidate libraries), and represents the genome as a $|l_x|$-size array $GV$, where the integer $p$ in position $q$ means that the object $q$ is contained into partition $p$. However, our purpose is the reduction of memory requirements for each application, therefore sometimes “cloning” an object in different libraries may help reducing the number of linked libraries. Unfortunately, the encoding schema above mentioned does not allow an object to be contained in more than one library.

A bit matrix encoding is therefore adopted, where the genome $GM$ for each library to split corresponds to a matrix of $k$ rows and $|l_x|$ columns, where $gm_{i,j} = 1$ if the object $j$ is contained into cluster $i$, 0 otherwise. Clearly, the presence of the same object in more libraries is indicated by more “1” on the same column.

Instead of randomly generating the initial population (i.e., the initial libraries), the GA is initialized with the encoding of the set of libraries obtained in the previous step.

The fitness function has been conceived to balance four factors:

1. The number of inter-library dependencies at a given generation;

2. The total number of objects linked to each application that, as said, should be as small as possible;

3. The size of the new libraries; and
4. The feedback given by the developers.

The first factor, the \( DF(g) \) (Dependency Factor) is defined as:

\[
DF(g) = \sum_{i=0}^{k} \sum_{j=i+1}^{k-1} md_{i,j} \delta(G[i], G[j])
\]  

(6.6)

where

\[
\delta(G[i], G[j]) = \begin{cases} 
0 & G[i] = G[j] \\
1 & G[i] \neq G[j] 
\end{cases}
\]

The second factor is the PR shown in the equation (6.5). The third factor, the \( SF(g) \) (Standard Deviation Factor) can be thought of as the difference between the initial library size standard deviation and the actual (at the current generation) standard deviation. Without taking into account the last item, it could happen that the GA, in the attempt to reduce dependencies, groups a large fraction of the objects in the same library, negatively affecting the PR. A similar factor was also applied in [13]. Given \( S_0 \) the array of library sizes for the initial population, and \( S_g \) the same for the g-th generation:

\[
SF(g) = |\sigma_{S_0} - \sigma_{S_g}|
\]  

(6.7)

The fourth factor keeps into account the developer feedback. After a first execution of GA, without considering this factor, developers are asked to provide a feedback on the proposed new libraries. The result of developer’s feedback is a bit matrix \( FM \) (Feedback Matrix), having the same structure of the genome matrix, and incorporating changes developers suggested with respect to libraries proposed by GA.

After this feedback, the GA is run again keeping into account, this time, the FF (Feedback Factor), accounting the difference between the genome and the FM:

\[
FF = \sum_{i=1}^{k} \sum_{j=1}^{k} |gm_{i,j} - fm_{i,j}|
\]  

(6.8)
In other words, the FM counts each time there is a difference between the genome and the solution proposed by developers.

Overall, the fitness function $F$ is defined as:

$$F(g) = DF(g) + w_1 \cdot PR(g) + w_2 \cdot SF(g) + w_3 \cdot FF(g)$$  \hspace{1cm} (6.9)

where $w_1$, $w_2$ and $w_3$ are real, positive weighting factors for the PR, SF and FF contribution to the overall fitness function. The higher is $w_1$, the smaller will be the overall number of objects linked by applications; on the other hand, increasing too much $w_1$ decreases dependency reduction. Similarly, the higher is $w_2$, the more similar will be the result to the starting set of library, while an excessively higher $w_2$ could not allow a satisfactory dependency reduction. Finally, $w_3$ should be properly sized to weight the influence of developer’s feedback. As explained, before asking developers for a feedback, a preliminary run of the GA must be performed with $w_3 = 0$.

As stated in (6.9), the fitness function is multi-objective [40, 41, 42]. Notice that, since the aim is to give maximum priority to dependency reduction, the DF weight is set to 1. Successively, $w_1$, $w_2$ and $w_3$ are selected using a trial-and-error, iterative procedure, adjusting them each time until the DF, PR, SF and FF obtained at the final step were not satisfactory. The process is guided by computing each time the average values for DF, PR, SF, and FF, and by plotting their evolution, to determine the 3D space region in which the population should evolve.

The crossover operator adopted is the one point crossover: given two matrices, both are cut at the same random column, and the two portions are exchanged (Figure 6.6a). The mutation operator works in two modes:

1. Normally, it takes a random column and randomly swaps two bits: this means that, if the two swapped bits are different then an object is moved from a library to another (Figure 6.6b); or

2. With probability $p_{clone} < p_{mut}$, it takes a random position in the matrix: if it is zero and the library is dependent on it, then the mutation operator clones the object into the current library (Figure 6.6c).
Noticeably, the cloning of objects increases both PR and SF, therefore it must be minimized. The GA activates the cloning only for the final part of the evolution (after 66% of generations in the case study). The mutation strategy favors dependency minimization by moving objects between libraries; then, at the end, it attempts to remove remaining dependencies by cloning objects. Obviously, at the end of the miniaturization process, cloned objects should be factored out again: if, for example, objects $o_a$ and $o_b$ are contained in both $l_i$ and $l_j$, then $o_a$ and $o_b$ should be moved into a third library from which $l_i$ and $l_j$ depend on.

Finally, the Lock Matrix accounts for a further, stronger level of feedback: this matrix gives to developers the possibility of locking an object in a cluster, when they strongly believe that object should belong to that cluster.

Given this, if a column of the Lock Matrix contains at least a not-null item, then the mutation operator does not perform any action bringing a genome in a inconsistent state with respect to the Lock Matrix. In other words, if the Lock Matrix indicates that object $j$ belongs to cluster $i$, any mutation removing object $j$ from cluster $i$ is avoided. It is worth noting that the crossover operator simply mixes columns from two genomes, thus it cannot perform any action in contrasts
with respect to the Lock Matrix. Similarly, a cloning mutation duplicates an object, without removing it from the cluster(s) where it was locked, thus even in this case no particular inconsistency check is necessary.

The population size and the number of generations are chosen by an iterative procedure, doubling both each time until the obtained DF, PR and FF were equal to those obtained at the previous step.

GA suffers from slow convergence: to improve performances, the GA was hybridized with hill climbing techniques. As reported in Section 6.3.2.1, hill climbing may be applied on individuals of the last generation, or on the best individual(s) of each generation. It has been experienced that, for this particular optimization problem, applying hill climbing only to the last generation does not significantly improve performances nor results. On the contrary, applying hill climbing to the best individuals of each generation makes the GA convergence significantly faster. In particular, hill climbing has been applied by repeatedly moving a random object from its source cluster to another cluster, and accepting the change only if the new genome increased the fitness function.

6.4.8 Identification of new Libraries

Due to its evolution, a software system tends to contain objects that, even if used by a common set of applications, are not contained into any library. Their identification and organization into libraries should be therefore desirable. The factoring process is quite similar to that described in the previous section. In particular, a MU matrix is built on a subgraph of the use graph obtained removing all existing libraries from it. Then, a first set of new candidate libraries is built by analyzing the dendrogram and the Silhouette statistics. These libraries are then refined with the aid of GA and developer’s feedback.

6.4.9 Library Remodularization using Dynamic Information

As described at the beginning of this chapter, the library remodularization will strongly reflect the way modules interact together during execution, if such remod-
Figure 6.7: The dynamic miniaturization process.

...ularization is based on dynamic information, i.e., on information obtained from execution traces.

The process proposed in Section 6.4.6 can be modified as shown in Figure 6.7. The software system undergoes two kind of analysis:

1. Static analysis, to obtain the MD and MU; and

2. Dynamic analysis (the distributed architecture used to collect traces is shown in Section 6.5.1).

Then, static and dynamic information are combined and, finally, the same steps described in Section 6.4.6 are performed.

Since, as shown in Figure 6.7 and as as it will be clearer later, we define new MD and MU as composed of static and dynamic information, from this point the static matrices will be referred as SMD (Static Matrix of Dependencies) and SMU (Static Matrix of Uses).

The dynamic information is accounted by two matrices:

1. DMU, where each item $d_{mu_{i,j}}$ is the frequency of uses of object $j$ by application $i$. The frequency is computed dividing the number of times application $i$ uses object $j$ by the total number of uses of library objects by all applications; and
2. DMD, where each item $dmd_{ij}$ is the frequency of accesses of library object $i$ to library object $j$. The frequency is computed as in matrix DMU.

To have at least a broad idea of the effectiveness of the trace extraction process (and therefore of the test suite) two Coverage Ratios, CR(DMU) and CR(DMD), are defined. In particular, CR(DMU) (Coverage Ratio of DMU) is defined as:

$$\begin{align*}
total &= 0 \\
covered &= 0 \\
\forall \text{ application } a_x \\
\forall \text{ object } o_y \\
&\quad \text{if } mu_{x,y} <> 0 \text{ then} \\
&\quad \quad total = total + 1 \\
&\quad \text{if } dmu_{x,y} <> 0 \text{ then} \\
&\quad \quad covered = covered + 1 \\
&\quad \text{end if} \\
&\quad \text{end if} \\
CR(DMU) &= covered \times 100 / total
\end{align*}$$

and CR(DMD) (Coverage Ratio of DMD) is computed similarly.

To combine static and dynamic information SMU and SMD not-null values were replaced by the lowest frequency contained respectively in the DMU and DMD matrices, assigning to static dependencies the meaning of “rare events”.

Given this, we can define the new MU and MD matrices, accounting for overall contribution, where:

$$\begin{align*}
mu_{i,j}(\lambda) &= \lambda \times smu_{i,j} + (1 - \lambda) \times dmu_{i,j} \\
md_{i,j}(\lambda) &= \lambda \times smd_{i,j} + (1 - \lambda) \times dmd_{i,j}
\end{align*}$$

(6.10)  (6.11)

Varying $\lambda$ it is possible to change the influence of static and dynamic information: in particular, for $\lambda = 1$ we obtain exactly the same static model described in
Section 6.4.6, while for \( \lambda = 0 \) the model is merely dynamic.

From this point, the proposed process in quite similar to what proposed in the previous section: after determining the optimal number of clusters \( k \) applying the Silhouette statistics and performing a preliminary clustering using agglomerative-nesting clustering, the new library are refined using GA.

The same PR measure of the miniaturization process performances, (although having a different meaning) can be used for the new approach. Let \( k \) the number of clusters \( l_{x_1}, \ldots, l_{x_k} \) obtained from a library \( l_x \). Then, the Probabilistic Partitioning Ratio \( PPR_x \) can be defined as:

\[
PPR_x(\lambda) = 100 \times \sum_{i=1}^{m} \frac{\sum_{j=1}^{k} |l_{x_j}| \ast m_{u_{i,x_j}}(\lambda)}{|l_x| \ast m_{u_{i,x}}(\lambda)}
\]  

(6.12)

where \( |l_x| \) is the number of objects archived into library \( l_x \).

The PPR has a slightly different interpretation with respect to the PR: the new algorithm tries to cluster together objects having high probability to be used by the same application(s), to avoid that an application linked objects unlikely to be used (dynamically loading them at run-time only when needed). Clearly, being MU function of \( \lambda \), it is possible to obtain different PPR indices having different static and dynamic contributions.

The multi-objective fitness function is the same of equation (6.9). However, the DF and PR have, in this context, a different meaning. In fact, the fitness function tries to minimize, at the same time:

1. The probability (not just the number) of inter-library dependencies (expressed by DF); and

2. The number of objects linked by applications but not frequently used (expressed by the PPR that, in the new fitness function, replaces the PR).

The genetic operators adopted are exactly the same described in Section 6.4.7.1.
6.5 Tool Support

To support the library reorganization and miniaturization process, different tools are needed. In particular, the following tools have been conceived:

- *The application identifier*: that, using the nm Unix tool, identifies the list of object modules containing the main symbol;

- *The graph extractor*: also based on the nm tool, that produces the System Graph, the Use Graph, and the Dependency Graph. The graph extractor also exports data in .DOT format [43], to allow visualization and analysis using the Dotty [44] graph visualization tool;

- *The unused symbol identifier*: it produces, for each library, the list of the symbols (and, for each one, the object containing it) not used by any application or library;

- *The circular dependency identifier*: it produces the list of all circular paths among libraries;

- *The duplicated symbol identifier*: it identifies the list of duplicate-defined external symbols. It is used in conjunction with the metric-based clone detector (see [45] for details) and with the dependency graph extractor to minimize the presence of clones inside libraries; and

- *The number of clusters identifier*, implementing the Silhouette statistics. In particular, implementations available in the cluster package of the R Statistical Environment [46, 47] have been used;

- *The library miniaturization tool*: it supports the process of splitting libraries in smaller clusters. The cluster analysis is performed by the agnes function available under the cluster package of the R Statistical Environment;

- *The GA library refiner*: implemented in C++ using the GAlib [48];
- **The developer's feedback collector:** it is a web application that allowed developer posting on our web site a feedback on the produced libraries;

- As in Chapter V, analysis performed with CA relied on the tool *concepts* [49].

### 6.5.1 Architecture for Collecting Dynamic Information

![Architecture for Collecting Dynamic Information](image)

Figure 6.8: a) Pipe architecture of the trace compressor. b) Deployment diagram of the trace collector system.

To obtain dynamic dependency information from a software system, it is necessary to instrument its code so that, during application execution, the following information is gathered for each function call:

1. The name of the application using the function;
2. The caller function and the name of the object module containing it; and
3. The called function and the name of the object module containing it.

Once an instrumented version of the software system to analyze is available, gathering traces presents two kinds of problems: the trace compression and the trace collection from a distributed execution environment.

The first problem is due to the enormous number of functions called, even for performing a simple task. To avoid trace database rapidly becoming unmanageable,
a pipe-based compressor (its architecture is shown in Figure 6.8-a) has been implemented. The instrumented program writes traces to a FIFO queue via thread safe-primitives (avoiding synchronization problems with other writing processes) rather than to a file. Then, traces are read from a compressor that performs two levels of compression:

1. RLE (Run Length Encoding) compression, replacing sequences of the same trace (generated, for example, by iterative or recursive calls) with a weighted call; and

2. Gzip compression.

Multiple applications run at the same time on several machines of different customers (i.e., the subjects from which traces were collected), and network connectivity may not be always present or the communication bandwidth may be devoted to higher priority processes. This leads to a distributed architecture, (shown in Figure 6.8-b) where traces are firstly stored in a temporary local database, then transferred to a centralized database.

Being customer's machines and trace analysis machines geographically distributed in different sub-networks, firewalls could deny communication through sockets or through middlewares such as CORBA (Common Object Request Broker Architecture). In this case the best solution is represented by web services. Communication is based on SOAP (Simple Object Access Protocol) protocol over HTTP (that traditionally operates on the TCP port 80, usually not filtered by firewalls). Periodically, a web service client sends locally collected traces to the remote trace collector web service, that stores them into a centralized database. A locally unique id (including a time stamp) is assigned by the collector to each client, ensuring that traces of different running instances were not mixed.

To effectively miniaturize software system libraries, it is essential that traces were obtained executing system applications in an environment as close as possible to the target profile of use. Moreover applications should be exercised in all (most of) their functionalities, considering all categories of inputs and also special situations.
In other words, a functional testing based on category partitioning [50] should be performed. For this purpose, results presented in this section were computed from dynamic information obtained combining both traces obtained performing a category partitioning test, and traces obtained letting users interact with the system for some (three-four) days.

To perform trace generation, collection and analysis, the following tools have been implemented:

- **The code instrumenting tool**, that wraps a C/C++ compiler (i.e., gcc/g++ in our case), by means of a recursive descendant hand-coded parser, and inserts probes into the (pre-processed) source code;

- **The communication library**, linked to the instrumented application to support local communication with the uncompressed trace queue (see Figure 6.8-b);

- **The trace compressor**, that reads from the FIFO queue traces written by the instrumented application, ad compresses them using RLE and Gzip;

- **The web service client**, that periodically flushes the local trace database, sending traces to the trace collector web service;

- **The trace collector web service**, that collects traces from client machines and stores them into the centralized trace database; and

- **The trace analyzer**, that extracts DMD and DMU matrices from the collected traces.

### 6.6 Applying the Miniaturization Process: Case Study

This section presents the results obtained applying to GRASS the reorganization process described in Section 6.4.
6.6.1 GRASS

As said in the introduction, GRASS is a large open source GIS. The GRASS CVS (Concurrent Versions System) development snapshot of April 5, 2002, downloadable from http://grass.itc.it was used as a case study. Its characteristics are summarized in Table 6.2.

<table>
<thead>
<tr>
<th>Pre-existing libraries</th>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library objects</td>
<td>921</td>
</tr>
<tr>
<td>Libraries</td>
<td>53</td>
</tr>
<tr>
<td>Applications</td>
<td>517</td>
</tr>
<tr>
<td>C source files</td>
<td>7107</td>
</tr>
<tr>
<td>C KLOCs</td>
<td>1014</td>
</tr>
</tbody>
</table>

Table 6.2: GRASS key characteristics.

Supported platforms at the date of writing comprise Linux/PC, SUN, HP/UX, MacOSX, MS-Windows/Cygwin, iPAQ/Linux and others. GRASS modules (commands) are organized by name, based on their function class (display, general, imagery, raster, vector or site, etc.). The first letter refers to the function class, followed by a dot and one or two other words, again separated by dots, describing the specific task performed by the module.

GRASS modules are invoked within a shell environment (or from the graphical user interface). The GRASS parser is a collection of subroutines which allow the programmer to define options (parameters) and flags that make up the valid command line input of a GRASS command.

The GRASS modules are linked against an internal “front.end”. The “front.end” module will call the interactive version of the command if there are no command-line arguments entered by the user. Otherwise, it will run the command-line version. If only one version of the specific command exists (for example, if there is only a command-line version available) the existing command is executed. Code parameters and flags are defined within each module. They are used to ask user to define map names and other options.

GRASS provides an ANSI C language API with several hundreds of GIS functions
which are utilized in the GRASS modules, from reading and writing maps to area
and distance calculations for georeferenced data as well as attribute handling and
map visualization. Details of GRASS programming are covered in the “GRASS 5.0
Programmer’s Manual” [51]. This programming API are organized as follows (typical
function name prefixes for related library functions are listed in squared brackets):

- **GIS library**: database routines (GRASS file management), memory manage-
  ment, parser (parameter identification on command line), projections, raster
data management etc. [G_], e.g., G_read_raster_row();
- **vector library**: management of area, line, and point vector data [Vect_, V2_,
dig_], e.g., V2_read_line();
- **image data library**: image processing file management [I_], e.g., I_georef();
- **site data library**: site data management [G_sites_], e.g., G_site_new_struct();
- **display library**: graphical output to the monitor [D_], e.g., D_new_window();
- **raster graphics library**: display raster graphics on devices [R_], e.g., R_open_driver();
- **segment library**: segmented data management [segment_], e.g., segment_get();
- **vask library**: control of cursor keys etc. [V_], e.g., V_ques();
- **rowio library**: for parallel row analysis of raster data [rowio_], e.g., rowio_get().

### 6.6.2 Handling Unused Objects

Out of 921 objects composing libraries, 89 were not used by any application, nor
by other libraries. It is highly desirable that those objects would be organized into
a separate cluster, thought of as a sort of repository to be “frozen” for future uses.
A deeper analysis revealed that some functions contained into unused objects wrap
lower level GRASS functions (e.g., db_create_index) standard library/system call
functions (e.g., scan_db1, scan_int, whoami), and, in general, provide some simple
functionalities using lower level functions (e.g., datetime_is_same, that compares
two DateTime structures. An interesting example (see also Section 6.6.5) is the library libdbmi: out of 97 objects, 19 were not used at all. In all cases, the unused functions correspond to one or more wrapped, lower level functions, that have been directly used by applications.

6.6.3 Removal of Circular Dependencies among Libraries

Three cases of circular dependencies among libraries were found. The first dependency was between libstubs.a and libdbmi.a. In particular, it was discovered that libstubs.a required one symbol, located inside the error.o module, (contained in libdbmi.a). On the contrary, libdbmi.a required 27 symbols from libstubs.a. The obvious solution was to move error.o into libstubs.a: this required moving in that library also the module alloc.o, since it depends from error.o.

The second circular dependency was found between libgis.a and libcoorcnv.a. In particular, libgis.a required three symbols from libcoorcnv.a, symbols located in the module datum.o (while the inverse dependency involved 13 symbols). Moving datum.o into libgis.a resolved the problem.

Finally, circular dependencies were found between libvect.a and libdig2.a. It involved 13 symbols in a direction, 31 in the other, symbols located in several different objects. The links present in the dependency graph excluded the possibility of resolving circular dependencies simply moving (or duplicating) objects. The decision taken (supported from system’s developers) was to initially merge the two libraries (in effect designed to work together) and then try to split again the new library (see Section 6.6.5).

6.6.4 Identification of Clones

The search for duplicated symbols followed by clone detection was performed at two different levels of the software system architecture: within libraries and on the whole system. In the first case, clone detection devoted to miniaturization purposes; in the second case, the objective was to identify portions of duplicated code that can be potentially organized into new libraries.
Table 6.3 reports results obtained from clone analysis: the total number of functions analyzed, the number of clone clusters [52] detected, the number and the percentage of cloned functions. Finally, clones were computed filtering out the shortest functions: for example, two functions that simply return a value should not be considered as clones. Results were presented considering two thresholds: functions longer than five and ten LOCs.

<table>
<thead>
<tr>
<th></th>
<th>Total # of Functions</th>
<th># of Clone Clusters</th>
<th># of Cloned Functions</th>
<th>% of Cloned Functions</th>
<th>Threshold (LOCs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>22229</td>
<td>2019</td>
<td>5789</td>
<td>26.0%</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1404</td>
<td>3641</td>
<td>16.38%</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Within Libraries</td>
<td>5271</td>
<td>72</td>
<td>180</td>
<td>3.41%</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>101</td>
<td>1.92%</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Outside Libraries</td>
<td>16958</td>
<td>1817</td>
<td>4974</td>
<td>29.33%</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1290</td>
<td>3288</td>
<td>19.27%</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Libraries vs. Outside</td>
<td>22229</td>
<td>130</td>
<td>635</td>
<td>2.86%</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>73</td>
<td>272</td>
<td>1.22%</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Results of clone detection.

As shown, the overall percentage of clones is not negligible (26%), even considering only functions longer than five LOCs (16.38%). Data in Table 6.3 suggests a potential reduction up to 17% in the number of the functions; clearly, the actual reduction rate will be lower due to false positives. The number of clones contained inside libraries is low, indicating that developer factored functions/objects accurately, avoiding duplicates. Finally, investigation was made about the set of clones between libraries and objects outside libraries, a situation where there could often be the possibility of a refactoring.

The analysis of clones inside libraries revealed an interesting situation: 16 symbols of library libortho, were cloned across libimage_sup, libmath and libtrans. Nine of the cloned functions were devoted to perform matrix algebra and, analyzing the Dependency Graph of libortho (see Figure 6.9), a subgraph composed of such functions was identified (i.e., the box on the right).

On the other hand, seven of the functions in the box on the left were cloned in libimage_sup. In particular, the entire structure enclosed in the rounded-dashed-box was replicated in that library. The decision taken was to split libortho in two libraries, corresponding to the two boxes in Figure 6.9:
1. A library (libmatrix) to handle matrices; and

2. A library (libcamera) to handle photogrammetric computations for aerial cameras.

Cloned functions contained in these two libraries were removed from libimage_sup, libgmath and libtrans.

![Diagram of library relationships]

Figure 6.9: Splitting library libortho.

Several “interesting” examples of clones were also found outside libraries. In particular, the r.mapcalc3 application contains four clusters of cloned, large functions (spanning from 27 to 59 LOCs). The first group contained functions called f_add, f_mul, f_sub, the second group f_cos, f_sin and f_tan, the third f_and and f_or, the fourth f_double, f_int, f_float. In all case, refactoring is clearly possible generalizing the operations and abstracting the types. We also found a function weisemberg_bingham cloned from shapiro_francia because (as from developer’s comment) the former statistics was not yet available in the system and it was temporarily replaced by the latter.

Finally clones were also found between applications and libraries. In most cases clones revealed to be part of legacy applications developed before the function was added into a library, and successively the application was never changed. A relevant (about 20%) fraction of these clones were discovered into the contrib sub systems, often developed by third-parties and therefore not always properly aligned with respect to the rest of the system.
6.6.5 Library Miniaturization

Miniaturization was performed on libraries composed of a large number of objects (see Table 6.6.5) following the process described in Section 6.4.6 and depicted in Figure 6.5. libproj was not split, as suggested by developers, in that it is currently under development by a different team. As explained in Section 6.6.3, libvect-new library was obtained merging libvect.a and libdiag2.a.

<table>
<thead>
<tr>
<th>Library</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgis</td>
<td>184</td>
</tr>
<tr>
<td>libdbmi</td>
<td>97</td>
</tr>
<tr>
<td>libproj</td>
<td>119</td>
</tr>
<tr>
<td>libvect-new</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 6.4: GRASS largest libraries.

First and foremost, Silhouette statistics was used to determine the optimal number of clusters for each library. Values of the statistics, for different number of clusters, are plotted in Figure 6.10. Given this, it was decided to split libgis into four clusters (instead of the six proposed in [20], libvect-new and libdbmi into three clusters. It is worth noting that, while for libgis the number of clusters was chosen in correspondence of the Silhouette maximum, for the other two libraries a compromise was pursued between maximizing the Silhouette and avoiding excessive fragmentation.

Subsequently, a preliminary clustering was performed and, then, results were refined with a first execution of GA, without considering any developer’s feedback (i.e., setting $w_3 = 0$). Table 6.6.5 reports, for each library:

- The number of objects composing the library;
- The number of candidate libraries the original library has been split into, and the corresponding Silhouette statistics value;
- The number of inter-library dependencies and the PR before applying the GA; and
Figure 6.10: Silhouette statistics for different number of clusters.

- The number of inter-library dependencies and the PR after applying the GA.

<table>
<thead>
<tr>
<th>Library</th>
<th># of objects</th>
<th>Candidate Libraries (k)</th>
<th>Silhouette statistics</th>
<th>Before GA</th>
<th>After GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DF</td>
<td>PR</td>
</tr>
<tr>
<td>libgis</td>
<td>184</td>
<td>4</td>
<td>0.70</td>
<td>579</td>
<td>51%</td>
</tr>
<tr>
<td>libdbmi</td>
<td>97</td>
<td>3</td>
<td>0.78</td>
<td>237</td>
<td>35%</td>
</tr>
<tr>
<td>libvect</td>
<td>54</td>
<td>3</td>
<td>0.57</td>
<td>66</td>
<td>46%</td>
</tr>
</tbody>
</table>

Table 6.5: Results of the library miniaturization process before considering feedback ($w_3 = 0$).

As shown, GA reduced **libgis** dependencies from 579 to 26 keeping the PR almost constant (from 51% to 48%). A significant reduction of inter-library dependencies was obtained (from 237 to 4 for **libdbmi** and from 66 to 3 for **libvect**), also slightly reducing the PR (except that for **libdbmi**, where it increased to 46%).

The first produced set of new **candidate libraries** was shown to developers to collect their feedback. For **libgis** the manual analysis indicated that the first cluster should contain “utility” and “allocation” functions, the second “area” and “geodesic” functions, the third “color-related” functions and the fourth “raster” functions. For
libvect-new developer indicated that the first cluster should contain basic filesystem operations, and the other two all other functions (a specific distinction was not supplied in this case).

The feedback for libdbmi was quite different with respect to the other two libraries. In this case developer (and also a manual graph analysis on visualizations produced with Dotty) revealed that the solution suggested by the hierarchical clustering (i.e., the preliminary clustering done before applying GA) reflected programmer's way to conceive the library. In fact, as also reported in [20], the library was split into three clusters:

- A cluster (libdbmi-1) containing (19) unused objects;
- A cluster (libdbmi-2) containing (30) objects directly used by applications; and
- A "low-level" library (libdbmi-3), containing 48 objects, used only internally to libdbmi.

Figure 6.11 reports the layering structure of clusters extracted from libdbmi. To avoid circular dependencies, one object was moved from libdbmi-3 to libdbmi-1.

Clearly, when performing miniaturization activities on a large software system such as GRASS, a compromise should be pursued between having smaller, decoupled clusters (like those generated applying GA) and clusters that, even if not totally decoupled, are conceptually cohesive (i.e., they contain functions implementing closely-related tasks). In the latter case, memory optimization is even possible adopting, as said, dynamically loadable libraries.

Given this, it was decided to leave libdbmi clusters as they were after hierarchical clustering, and to perform a "second round" of GA clustering on libgis and libvect-new, considering, this time, also the FF. For sake of completeness results for libdbmi were also reported. By varying the $w_1$, $w_2$ and $w_3$ thresholds, different results were obtained. As shown in Table 6.6.5, it was never possible to pursue cluster complete decoupling and obtaining, at the same time, libraries very close to the structure proposed by developers (i.e., a small FF).
Table 6.6: Results of the second round of the library miniaturization process ($w_3 \neq 0$).

<table>
<thead>
<tr>
<th>Library</th>
<th># of objects</th>
<th>Candidate Libraries (k)</th>
<th>Before second round</th>
<th>After second round</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>FF</td>
<td>DF</td>
</tr>
<tr>
<td>libgis</td>
<td>184</td>
<td>4</td>
<td>203</td>
<td>26</td>
</tr>
<tr>
<td>libdbmi</td>
<td>97</td>
<td>3</td>
<td>97</td>
<td>4</td>
</tr>
<tr>
<td>libvect</td>
<td>54</td>
<td>3</td>
<td>72</td>
<td>3</td>
</tr>
</tbody>
</table>

The comparison of the first three columns with the last three highlights that, after the first GA round, the coupling between clusters was kept low. On the other hand, the libraries produced tend to have a meaning different to what intended by developers (this was highlighted by the high FF before second round). The second round of GA tried to reduce the FF; this however increased coupling. At this stage the developers may decide to produce meaningful libraries and reducing the memory requirements using dynamic-loadable libraries, or to obtain independent cluster, even if these clusters did not always group conceptually related objects.

The adoption of a hybrid GA approach, as mentioned before, did not allow to improve accuracy in that, increasing the number of generations and the population size, pure GA also converged to similar results. Noticeably, performing hill climbing on the best individuals of each generation produced a drastic reduction of convergence times. Comparing both strategies when the difference between values of the fitness function was below 10% highlighted that a hybrid strategy allowed to reduce, on average, the execution time of 43%. Convergence times for a Compaq Proliant$^{TM}$ (Dual Xeon$^{TM}$ 900 MHz processor, 2MB Cache and 4GB of RAM) are reported in
Table 6.6.5.

<table>
<thead>
<tr>
<th>Library</th>
<th>Pure GA Fitness Function</th>
<th>Time (sec.)</th>
<th>Hybrid GA Fitness Function</th>
<th>Time (sec.)</th>
<th>Fitness % Diff.</th>
<th>Time % Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgis</td>
<td>3112</td>
<td>913</td>
<td>3229</td>
<td>4524</td>
<td>1%</td>
<td>49%</td>
</tr>
<tr>
<td>libdmi</td>
<td>77</td>
<td>509</td>
<td>83</td>
<td>190</td>
<td>7%</td>
<td>37%</td>
</tr>
<tr>
<td>libvec</td>
<td>195</td>
<td>96</td>
<td>198</td>
<td>41</td>
<td>3%</td>
<td>43%</td>
</tr>
</tbody>
</table>

Table 6.7: Performance comparison between pure GA and hybrid GA with hill climbing.

### 6.6.6 Extraction of new Libraries

The final step of the miniaturization process is devoted to analyze the *Use Graph* obtained removing all existing libraries, to investigate the existence of new candidate libraries: sometimes there may be groups of objects used by a common set of applications, but they have not yet been clustered into libraries. To perform this task, clustering was performed on objects used by, at least, two applications.

Results revealed the presence of four clusters, all located in the *orthophoto* subsystem. The number of dependencies between clusters was small, and it was possible to solve them simply moving between clusters a couple of objects. Besides, all clusters had a considerable number of dependencies to external objects (i.e., other objects belonging to their same set of applications). To eliminate these dependencies, it would be necessary to increase the size of each cluster of over 100%. This is clearly contradictory with respect to the principles the miniaturization process relies on. Consequently, it was decided not to cluster these objects into libraries. It is worth noting that this is not a negative result; on the contrary, this constitutes a quality indicator of the system: developer carefully created and maintained libraries.

### 6.7 Library Identification with CA: Case Studies

This section aims to show how CA can be used to identify libraries from object module dependencies. As explained in Section 6.4.6, the CA based approach has both pros (visualization of dependencies) and cons (a concept lattice rapidly becomes
unmanageable). The approach has been applied to the same software systems used in Chapter V for directory reorganization, i.e., mSQL and Samba.

A single candidate library was identified on mSQL 1.0.6: the objects libmSQL.o and net.o are used in all the executables. It is worth noting that such objects have been actually organized in a library by the original developers. Another candidate library was found within mSQL 1.0.6: in fact, mSQL_lex.o, mSQL_yacc.o, mSQL_proc.o, acl.o, regexp.o and mSQLdb.o are used by mSQLdump, relshow, mSQLadmin and mSQLd. However, developers did not build a library with that files even if it could be convenient: four out of seven executables used those objects.

In mSQL 2.0.8 three candidate libraries were identified; two of these, Lib1 and Lib2, are quite similar respectively to libmysql and liblite created by developers. mSQL 1.X and 2.X developers clustered regexp.o and regsub.o into libregexp.a; however, this library is needed only by one executable in mSQL 1.X and by three executables in mSQL 2.X. Thus, libregexp.a was not highlighted as a “candidate” library by the search heuristic. A complete comparison between extracted libraries and real libraries is shown in Table 6.7.

<table>
<thead>
<tr>
<th>RETRIEVED LIBRARIES</th>
<th>ACTUAL LIBRARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lib1</td>
<td>Lib2</td>
</tr>
<tr>
<td>net</td>
<td>config</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8: mSQL 2.0.8 retrieved libraries vs. real libraries (.o files).

Samba 1.9.08 developers did not create any library; however, two candidate li-
libraries could be identified. In particular,

- `util.o` is used by all applications and it may be obviously considered a library; and

- `pcap.o`, `params.o` and `loadparams.o` are used by `testprns` and `smbtatus`, so they could be clustered in a library.

![Diagram of Samba 2.0.5 objects lattice](image)

Figure 6.12: Samba 2.0.5 objects lattice.

Samba 2.0.5 was the most complex system among those analyzed. Figure 6.7 shows the concept lattice produced by applying the second step to such a software system: two candidate libraries could be identified, the first one of which is used by all executables. A comparison between retrieved libraries and libraries built by developers is shown in Table 6.7. It can be stated that:

- For `Lib2`, there is both a good precision and a good recall [53]; and
• For Lib1, while the precision is 100%, the recall is above 65%.

The Lib1 lower recall value is due to the fact that only the chosen files are linked by all the executables (16 files), disregarded files are not. The manual Lib1 analysis of dependencies revealed that almost each disregarded file was specific to a particular main, in other words, no discarded file was linked by more than the 30% of all executables. Files such as getsmbpass.o, interface.o and netmask.o appearing several times were moved by the approach from Lib1 to Lib2.

```
<table>
<thead>
<tr>
<th>RETRIEVED LIBRARIES</th>
<th>REAL LIBRARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lib1</td>
<td>Lib2</td>
</tr>
<tr>
<td>charconv</td>
<td>namequery</td>
</tr>
<tr>
<td>system</td>
<td>nmbib</td>
</tr>
<tr>
<td>kanji</td>
<td>nterr</td>
</tr>
<tr>
<td>util_file</td>
<td>smberr</td>
</tr>
<tr>
<td>signal</td>
<td>smbencrypt</td>
</tr>
<tr>
<td>spritef</td>
<td>smbdes</td>
</tr>
<tr>
<td>util_sock</td>
<td>util_sock</td>
</tr>
<tr>
<td>dcecalls</td>
<td>debug</td>
</tr>
<tr>
<td>debug</td>
<td>interface</td>
</tr>
<tr>
<td>util_str</td>
<td>netmask</td>
</tr>
<tr>
<td>username</td>
<td>getsmbpass</td>
</tr>
<tr>
<td>util</td>
<td>genrand</td>
</tr>
<tr>
<td>md4</td>
<td>fault</td>
</tr>
<tr>
<td>time</td>
<td>getsmbpass</td>
</tr>
<tr>
<td>charconv</td>
<td>interface</td>
</tr>
<tr>
<td>genrand</td>
<td>uid</td>
</tr>
<tr>
<td>genrand</td>
<td>replace</td>
</tr>
<tr>
<td>fault</td>
<td>ufe</td>
</tr>
<tr>
<td>getsmbpass</td>
<td>access</td>
</tr>
<tr>
<td>interface</td>
<td>bitmap</td>
</tr>
<tr>
<td>netmask</td>
<td>crc32</td>
</tr>
<tr>
<td>getsmbpass</td>
<td>util_ssid</td>
</tr>
<tr>
<td>interface</td>
<td>util_uniqnr</td>
</tr>
<tr>
<td>netmask</td>
<td>util_sec</td>
</tr>
<tr>
<td>getsmbpass</td>
<td>ss1</td>
</tr>
</tbody>
</table>
```

Table 6.9: Samba 2.0.5 retrieved libraries vs. real libraries (.o files).

### 6.8 Dynamic Miniaturization: Case Studies

This section reports results obtained applying the dynamic refactoring to Samba and MySQL libraries.
6.8.1 Case Study Description

**Samba** is a freely available file server. A detailed description of Samba can be found in Section 5.5. Traces for dynamic miniaturization were extracted during functional testing and four days of normal use. All Samba functions were exercised in their options, for example: changing passwords and managing users with `smbpasswd`; accessing and modifying Samba configuration with `testparm` and `swat`; and performing several file-transfer operations (getting files, putting files, removing files, creating/renaming/removing directories, etc.) both with `smbclient` and GUI (Graphical User Interface) clients such as Microsoft Windows Explorer™ or Gnome.

**MySQL** ([http://www.mysql.com/](http://www.mysql.com/)) is an open source, fast, multi-threaded, multi-user SQL database server, intended for mission-critical, heavily loaded production systems. MySQL is written using both C and C++, and can be compiled with several different C/C++ compilers. The power of MySQL is in its fastness: in order to pursue this objective, some advanced features (e.g., nested queries) are not available, while others (e.g., transactions) were introduced only in the latest version of the database server.

To obtain traces for dynamic miniaturization from MySQL, all its utilities were exercised as explained in Section 6.5.1. In particular: administrating databases, e.g., creating, flushing, removing databases; checking MySQL configuration; checking and compressing indexed sequential (ISAM) files; accessing to MySQL log files; importing/exporting tables from/to ASCII files; dumping a database; and, above all, performing several SQL operations from the interactive console.

As explained in Section 6.6, only the biggest libraries were split. Characteristics of the four systems analyzed are shown in Table 6.8.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Ver</th>
<th>KLOCs</th>
<th>Apps</th>
<th>Libs</th>
<th>Libs to re-factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>3.23.51</td>
<td>378</td>
<td>38</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>Samba</td>
<td>2.2.7</td>
<td>295</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.10: Dynamic miniaturization: case studies characteristics.
6.8.2 Case Study Results

The two Samba libraries were re-organized, trying to minimize dynamic dependencies between them, clustering together, at the same time, objects having a high frequency of use by a common set of applications. With the same criteria, the two largest MySQL libraries, libmysqlclient and libmysys, were decomposed in small chunks.

Not surprisingly, dynamic execution exploited a limited set of uses and dependencies: for Samba a \( CR(DMD) \) of 40\% and a \( CR(DMU) \) of 30\% were obtained. Indices were even smaller for both MySQL libraries: \( CR(DMD) \) of 25\% and \( CR(DMU) \) of 20\%. This supported the approach, described in Section 6.4.9, of combining static and dynamic information. On the other hand, it should be considered that different user execution profiles will lead to different coverage indices.

The proposed miniaturization process was performed for three different values of \( \lambda \): 0, 1 and 0.5, i.e., considering only dynamic information, only static information, and a combination of both.

The first step was to determine the optimal number of clusters: for \( \lambda = 1 \) and \( \lambda = 0.5 \), Silhouette statistics gave us the same optimal number of clusters, i.e. two for Samba, three for libmysqlclient and two for libmysys.

Results obtained combining both static and dynamic information are shown in Table 6.8.2. DF and PPR before and after applying GA are reported. This time DF is, as explained, a measure of the dynamic coupling between libraries. Static DF (i.e., the number of static dependencies after applying GA, indicated with \( DF(1) \)), are also reported, as well as static PPR (indicated with \( PPR(1) \)). It is worth noting that PPR values are very smaller with respect to PR ones: in other words, the algorithm tries to cluster objects having a high frequency of common use.

Table 6.8.2 highlights different results. First of all, the GA significantly (for Wilcoxon tests performed with significance level \( \alpha = 95\% \)) reduced the dynamic dependencies, leaving almost unaltered the PPR indices. The latter is not a negative aspect: PPR obtained, as it will be discussed below, are already good after hier-
archical clustering. On the other hand, GA tried to minimize the DF as much as possible, without significantly affecting the PPR. This allowed to create *dynamically independent* libraries, i.e., with a low probability of coupling.

Although, in this case, the objective of the fitness function was to minimize dynamic dependencies and to cluster together objects having high frequency of common use, it is interesting to analyze what happened to static parameters (PPR(1) and DF(1)). Clearly, as shown by values over 90% and, in some cases, close to 100%, the PPR(1) was not minimized by hierarchical clustering nor by GA. Again, this is not surprising: common, but rare use of objects by the same set of applications were not weighted, in terms of importance, as the most frequent use relationships.

About DF(1), Table 6.8.2 shows a significant reduction after applying GA. Resulting static dependencies (from 10 of *libmysys* to 37 of *libmysqlclient*) were not so small to allow (like in the static analysis performed in [21] and shown in Table 6.8.2) obtaining completely independent libraries, but they were minimized to reduce the coupling even for rare events.

After discussing obtained figures, results should be interpreted analyzing the meaning of the libraries obtained. Analyzing the two new *Samba* libraries it has been observed that, for example, the first library contained objects frequently used by daemons (*smbd* and *nmbd*), while the second library contained objects often used by administration utilities, such as *smbpasswd*. Similarly, *MySQL libmysys* library was split in a library containing general-purpose functions, often used by the SQL engine, and another library containing utility functions used by different *MySQL* applications. Finally, *libmysqlclient* was split in a library containing general-purpose functions (memory allocation, stream handling, etc.), a library containing utility functions for client applications, and a library for interfacing client applications with the database server.

Finally, miniaturization was performed considering the case of \( \lambda = 0 \), i.e., taking into account only dynamic links. Although DF(\( \lambda = 0 \)) factors were successfully minimized and PPRs were comparable (and even smaller) to those obtained for \( \lambda = 0.5 \), the decision taken was to discard these results. This because the number of static
dependencies after applying GA remained high, increasing the risk that a dependency not exploited by instrumentation will occur, forcing therefore an extra library to be loaded at run-time. In other words, results obtained for $\lambda = 0$ confirmed one of the most important lessons learned from this work: dynamic information allows an effective miniaturization, but it should be complemented with static information.

<table>
<thead>
<tr>
<th>System</th>
<th>Library</th>
<th># of objects</th>
<th># of objects</th>
<th>Silhouette statistics</th>
<th>Before GA DF (1)</th>
<th>Before GA PPR (1)</th>
<th>After GA DF (1)</th>
<th>After GA PPR (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samba</td>
<td>lib + libmb</td>
<td>77</td>
<td>2</td>
<td>0.71</td>
<td>106</td>
<td>72%</td>
<td>2</td>
<td>64%</td>
</tr>
<tr>
<td>MySQL</td>
<td>libmysqlclient</td>
<td>80</td>
<td>3</td>
<td>0.64</td>
<td>187</td>
<td>76%</td>
<td>7</td>
<td>70%</td>
</tr>
<tr>
<td>MySQL</td>
<td>libmysql</td>
<td>92</td>
<td>2</td>
<td>0.47</td>
<td>158</td>
<td>89%</td>
<td>1</td>
<td>66%</td>
</tr>
</tbody>
</table>

Table 6.12: Results for $\lambda = 0.5$.

6.9 Conclusions

This chapter presented the framework activities related to software miniaturization, as well as results from their application to an over 1 million LOCs software system such as the GRASS GIS. At the time of writing the application was successfully ported on a PDA (i.e., a CompaQ iPAQ). Given the size of the application and the available resources, a brute force automatic approach was not feasible: the developer’s suggestions revealed to be an essential component for the miniaturization process.

The framework allowed to remove several, physiological problems from the analyzed software. In particular, unused objects were identified and factored out; clones were identified and, especially for clone inside libraries, refactoring was performed. Cloning level outside libraries was not negligible, suggesting further activity of clone
refactoring. Besides, the cloning level inside libraries was low, except that for some situations such as those highlighted, and the cloning between libraries and rest of the system was, on most cases, due to third party applications.

A multi-steps library miniaturization process was proposed, where hierarchical clustering identified a first, sub-optimal solution then refined by GA. The proposed fitness function accounts for different factors: minimizing dependencies, the average number of objects linked by each application and, finally, the feedback given by the developers.

The framework allowed to effectively reorganize the structure of GRASS, reducing its memory requirements and improving its performances: the average number of library objects linked by each application was reduced of about 50%. Finally, the proposed framework can also be used as a quality metric profile: unused objects, clones, library coupling, library sizes and poor object organization are in fact significant quality indicators. For instance, the identification of no new libraries in GRASS indicated a careful design and a controlled evolution.

A CA-based factoring approach allow to visualize, on the concept lattice, useful information on object dependencies. However, such a lattice becomes unmanageable when dealing with large software systems.

By incorporating dynamic information into clustering and GA, libraries were optimized with respect to a given user profile. However, different profiles would have lead to different libraries, making the proposed mechanism appealing for generating customized software configurations. Combining static and dynamic information allowed to keep into account dependencies not exploited by software instrumentation.
Bibliography


[43] AT&T. Dotty directed graph editor (part of graphviz).


http://www.r-project.org.


http://lancet.mit.edu/ga/.

[50] T. Ostrand and M. Balcer. The category-partition method for specifying and
generating functional tests. Communications of the Association for Computing
Machinery, 31(6), June 1988.


in the Linux Kernel. Information and Software Technology (SCAM 2002 Special

[53] W. B. Frakes and R. Baeza-Yates. Information Retrieval: Data Structures and
CHAPTER VII

Traceability Link Extraction

7.1 Introduction

Traceability is the mechanism that allows to create links between and within software artifacts. The IEEE Glossary of Software Engineering [1] defines the traceability as:

"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match."

Pfleeger [2] made the distinction between horizontal traceability, i.e., the traceability between documents at the same level of the software life-cycle (e.g., the traceability between related requirements) and vertical traceability, i.e., the traceability between documents at different levels of the software life-cycle (e.g., the traceability between requirements and source code components).

It is not infrequent that, while source code evolves, documentation is not updated; maintaining consistency and traceability information between software artifacts is a costly and tedious activity frequently sacrificed due to the market pressure. Furthermore, since outsourcing has been widely adopted as a common practice, the people who developed or maintained a system may no longer be available. Indeed, often the only reliable source of information about a software system is the system itself.

Traceability creates links between and within software artifacts. The chapter
focuses on extracting traceability links between artifacts, and more precisely, areas of code and related sections of free text documents such as a requirement document, an application-domain handbook, a specification document, a set of design documents, or manual pages. Establishing traceability links between the free text documentation associated with the development and maintenance cycle of a software system and its source code can be helpful in a number of tasks [3]. A few notable examples are:

- **Program comprehension:** program comprehension is generally performed in bottom-up [4, 5] or top-down [6, 7] manner: in both cases traceability links reveals to be useful. In a top-down approach a high-level hypothesis can be related to source code, while in a bottom up approach traceability allows to align high-level concepts to source code or to aggregate source chunks in more abstract concepts;

- **Software maintenance:** given the functional requirements to be changed, traceability links allow to identify the source code to be accordingly maintained. On the other hand, it allows to align documentation with respect to changed source code;

- **Requirement tracing:** traceability links are useful to assess the completeness of an implementation with respect to the corresponding requirement, to devise complete test cases, etc.

- **Impact analysis:** traceability links allows to identify the deliverables affected by a given change; and

- **Reuse:** Tracing source code to free text document significantly helps to locate subsystems/components candidate for reuse.

Existing cognitive models share the idea that program comprehension occurs in a bottom-up manner [4, 5], a top-down manner [6, 7], or some combination of the two [8, 9, 10, 11]. They also agree that programmers use different types of knowledge ranging from domain specific knowledge to general programming knowledge [6, 12,
Programmers need to build a mental model of the software system, i.e., high-level abstractions preliminary to development or maintenance activity.

This work assumes that such a mental model exists and subsumes that programmers tend to process application-domain knowledge and to use high-level abstractions in a consistent way. Therefore, program item names of different code regions or words of a given text document related to the same concepts are likely to be the same or at least very similar. Under the above assumption, the knowledge of existing traceability links can be exploited modeling the mental map adopted by programmers to link high-level concepts present in textual documentation with source code regions.

In [3, 14] stochastic modeling and maximum likelihood classification were adopted to model programmers as stochastic communication channels. The main limitation of [3, 14] was the simplistic idea that concepts are captured by single words. To overcome such a limitation, the mathematical model was redefined to accommodate subsets of words; since the number of possible word sequences for any given text document is the power set of the words, we limited ourselves to sequences of adjacent words (for each document, the number of terms considered, i.e., single words plus pairs of adjacent words, is upper bounded by twice the number of words in the document itself). Once a subset of traceability links is available (i.e., couples high-level/low-level artifacts), the joint probability distribution of the text document (sequences of words) and a set of linked areas of code (program item names) are estimated together with the document words marginal probability distributions for any given link. The estimated probability distributions are used in a maximum likelihood classifier to score sequence of mnemonics extracted from a not yet classified area of code.

The approach was applied to recover traceability links in three different systems: _ALBERGATE, Transient Meter_ and a system managing the University library, called _Library Management_. Each case study in turn may be considered representative of different technologies and development approaches.

_ALBERGATE_ was developed in Java following a waterfall development model. _Transient Meter_ is a software developed in C++ while the _Library Management_
system was developed with Visual Basic; the latter systems (in particular *Transient Meter*) may be thought of as representative of industrial software developed with RAD (Rapid Application Development)) and a prototyping approach.

Industrial software developed with COTS components (e.g., database access components), automatically generated code (e.g., by GUI builders or report generators), and middleware (CORBA) poses new challenges to the traceability link recovery. This chapter presents a taxonomy of traceability recovery affecting factors, previously analyzed in [15]. The newly proposed equations overall improved performances measured as precision and recall [16] on the three systems.

The remainder of the chapter is organized as follows: related works are discussed in Section 7.2, the programmer’s cognitive model is presented in Section 7.3, then formalized in Section 7.4, extending the equations presented in [3, 14]. Section 7.5 proposes a taxonomy of the traceability links affecting factors, while Section 7.6 presents the process to filter the information to recover traceability links, and the tools developed to automate the process itself.

Case studies are presented in Section 7.7. Results are presented in Sections 7.7.2 and discussed in Section 7.8. Finally, Section 7.9 gives concluding remarks and outlines directions for future work.

### 7.2 Related Work

The issue of recovering traceability links between code and free text documentation is not yet well understood and investigated, and very few contributions were published in the past 20 years. A number of related papers have been published in the area of impact analysis. For example, Turver and Munro [17] assumed the existence of some form of ripple propagation graph, describing relations between software artifacts, including both code and documentation, and focusing on the prediction of the effects of a maintenance change request, not only on the source code, but also on the specification and design documents.

The requirement traceability problem has been investigated and discussed in several works: Gotel and Finkelstein [18] proposed empirical studies and showed the
motivations for which there is little progress in this area, and what is necessary to achieve improvements.

Boldyreff et al. [19] presented a method for the identification of traceability links of high-level domain concepts, using all available maintenance information, and starting from a different view of traceability respect to IEEE definition.

IBIS [20], TOOR [21], PRO-ART [22] and REMAP [23] are few examples of software development tools able to build and maintain traceability links among various software artifacts. RECON [24] is a tool developed starting from Software Renaissance, a dynamic analysis technique proposed in [25] and [26]. Ramesh and Jarke in [27] followed an empirical approach for define, validate and apply a reference model of traceability based on the previous experiences with the deployment of REMAP. However, these tools are focused on the development phase; furthermore, they require human intervention to define links, or they force the adoption of naming conventions.

Maarek et al. [28] introduced an IR (Information Retrieval) method for automatically assembly software libraries based on a free text-indexing scheme. The method uses attributes automatically extracted from natural language written IBM RISC System/6000 AIX 3 documentation to build a browsing hierarchy, which accepts queries expressed in natural language. GURU [29] is a system that automatically identifies indices, by analyzing the natural-language documentation, in the form of manual pages or comments, usually associated with the code. This system has been applied to construct an organized library for AIX utilities.

Several software reuse environment use IR to index and retrieve the reusable assets. The RLS [30] system extracts free-text single-term indices from comments in source code files, looking for keywords such as “author”, “date created”, etc. REUSE [31] as an IR system, which stores software objects as textual documents in view of retrieval for reuse. Similarly CATALOG [32] stores and retrieves C components, each of which is individually characterized by a set of single-term indexing features automatically extracted from natural language headers of C programs.

Maletic and Marcus [33] presented a system called PROCSSI, that uses IR tech-
nique to identify semantic similarity between pieces of code, proving that this semantic similarity measure, used to cluster software components, is an helpful support in the comprehension task.

Antoniol et al. [34] presented a method to establish and maintain traceability links between code and free text documents. The method exploits probabilistic IR techniques to estimate a language model for each document or document section, and applies Bayesian classification to score the sequence of mnemonics extracted from a selected area of code against the language models.

The same method was applied in [35], to recover traceability links between the functional requirement and the Java source code, extending and validating the previous results on a more complex and difficult case study. The investigation was then extended in [3, 36] to vector space models, to compare different models families and to assess the relative influence of affecting factors. Latent semantic approach [34] and IR approach were compared in [37].

The case study, the equations, the approach and the process proposed in this chapter differ in several aspects from those of papers mentioned above. More commonalities can be found with [14], where a simplified version of the programmer cognitive model, in which each concept is associated to a single word, was proposed and applied to ALBERGATE. Studies on factors affecting traceability were performed in [15], and the process aiming to remove these factors was applied to Transient Meter.

7.3 Programmer’s Cognitive Model

The traceability recovery approach described in this chapter relies on the hypothesis that programmers tend to name identifiers (variables, functions, etc.) consistently with their mental model, the domain, the problem knowledge and the naming convention adopted in the high-level documents (i.e., requirements, design, use cases, etc.).

This means that a concept (i.e., an object of the application domain, a property, a relation, an actor) is firstly referred consistently in the high-level documents, such
as the SRS (Software Requirements Specifications) [38], and then in the low-level artifacts. Consistency is promoted by development practice and standards. For example, [38] enforces the consistency and the non-ambiguity, and requires a “keywords and acronyms” paragraph defining terms that are associated to concepts, terms used consistently in the document.

Programmers map problem and domain concepts in software artifacts consistently: low-level artifact terms reflect programmer knowledge as well as high-level artifact terms. In other words, the terms appearing in different parts of software artifacts at a given stage of the software life cycle (e.g., the design documents or the code components) bound to a given concept are likely to be similar, while potentially different from the word used in higher-level documents (e.g., SRS) produced in early phases.

Consider, for example, a requirement from a library management application:

*It must be possible to insert a new book into the library. The user must insert the following information:*

- **The title** of the book;

- **The names of the authors**; and

- **The number of pages** of the book.

*After confirming the data inserted, data are stored into the database. In case of abort, nothing shall happen.*

In a OO implementation, programmers will map this requirement into different classes and among others, a class containing the book data structure (an excerpt of such a class is shown in Figure 7.1-a), and a class implementing a new book insertion form (Figure 7.1-c and Figure 7.1-b). Let us highlight, for instance, these different situations:

1. The same terms appear in the requirements and in the code (**title**);

2. The identifiers **Add** and **Canc** are associated, respectively, to the words **confirming** and **abort**;
3. A different term (\texttt{Npages}) is associated with a word (\texttt{pages}) used in the requirement, and it is used consistently in the code;

4. The requirement words \texttt{number} and \texttt{pages} may be mapped to the identifier \texttt{Npages}. This means that, in some cases, programmers tend to map a set of words, corresponding to a domain or problem concept, into a composite program item name; and

5. The requirement identifier \texttt{authors} is mapped, inconsistently, into two different terms \texttt{Auths} and \texttt{Authors}.

![Figure 7.1: How the programmer can map concepts into code.](image)

The key idea is that, as in point 4, several words of a requirement contribute to create a single program item name. However, it is not unlikely that a concept/object may be often described by a single word or a couple of adjacent words. Although some previous contributions suggest that a concept should always be expressed by a single word (nouns, see [39]), and thus, traceability recovery should focus on single
words of high-level documents, we discovered a non-negligible fraction of concepts mapped into multiple words (see Section 7.7.2).

The above hypothesis can be re-formulated as follows: when reading the requirements, programmer tends to use a “sliding window” spanning a limited number of words (not counting stop words i.e., articles, prepositions) to create program item names.

A challenging issue is to choose the window size, i.e., the maximum number of adjacent words retaining a concept. One may decide to consider the power set of the words in a requirement. However, this is infeasible, and moreover, in practice, a concept is often well localized in the text. Thus, a sliding window of few words should be considered. This is also in agreement with the general consensus concerning the short-term memory and the ability to concurrently process different information chunks (about seven, according to [40, 41]).

Another issue is to differently weight single words and words sequences; it is possible that concepts are retained in a single word with a higher probability than in multiple words. In our model, this is formalized by different weights associated to terms of different size (see Section 7.4). Details on how calibrating such weights can be found in [42].

Figure 7.1-b highlights a class containing identifiers automatically generated by a tool (in this case the Java class, implementing a frame has been drawn using Borland JBuilder™). These identifiers are not tied to the problem domain, nor to the programmer experience or mental model. They represent one of the factors negatively affecting traceability and traceability recovering processes.

Finally, it has been experienced that, if identifiers associated with a concept in requirements are not used consistently (Auths and Authors associated to the word authors), the performances of a recovery process are likely to deteriorate as the percentage of this inconsistencies increases.
7.4 The Mathematical Model

Stochastic models are used extensively in several areas: automatic speech recognition, machine translation, spelling correction, text compression, etc. [43, 44, 45].

In the software development and maintenance life cycle, programmers may be thought of as stochastic channels transforming a high-level text document into high-level abstractions and knowledge and finally into observations i.e., chunks of code. Development or maintenance activity is modeled by a stochastic channel where the sequence of words (a high-level text document) \( W_k \), generated by a source \( S \) with the \textit{a-priori} probability \( Pr(W_k) \), mapped by the programmer into an abstraction \( A_k \), is transformed into the sequence of observations \( O \) with probability \( Pr(O | W_k) \), more precisely \( Pr(O | A_k) \). In the traditional pattern recognition applications, \( O \) could represent, for example, the acoustic signal produced by uttering \( W_k \), the translation of \( W_k \) from Italian to English, or a typewritten version of \( W_k \) with possible misspellings. In the software engineering domain, to model programmer activities, \( W_k \) may represent a functional requirement or a maintenance change request, while \( O \) may correspond to the program item names i.e., the mnemonics for code identifiers chosen by the programmers. Notice that any word of the \( W_k \) document (as well as other sources of knowledge) will possibly influence the entire observation \( O \).

As stated in Section 7.3, the method’s underlying assumption is that programmers tend to build code identifiers (either variable of function name) processing with a set of unknown rules, domain and problem knowledge. Those unknown rules are not expected to change sharply over the time and from one programmer to the other in that rules depend on the corporate culture, on adopted coding standard and the programmer skills. Hence, the program item names of different code regions are likely to share semantic meaning with the higher-level abstractions and concepts, or at least they have been created consistently applying the set of unknown rules. Thus, program item names of different code regions related to a given high-level abstraction or concept (possibly, corresponding to a given text document) are likely to be the same, or at least very similar.
We can assume that concepts embodied into a fragment of a high-level document can be represented and triggered in a high-level abstraction by terms appearing in the text fragment. A term corresponds to:

1. A single word (i.e., the unigrams [3, 14]);
2. Two or more consecutive words; and
3. More generally, a subset of the words belonging to the text fragment.

Given a document $W_k$ its abstract representation $A_k$ is modeled by the superposition of terms $t_{k,i}$:

$$A_k = \bigcup_i \{t_{k,i}\}$$

In general, the difficulty to precisely locate concepts and words triggering concepts leads to a combinatorial explosion. Thus, in the following, we will disregard the third item and assume that concepts are represented and triggered by a single word or, by at most two adjacent words (also called bigram in the following). A further complication arises when $Pr(A_k)$ needs to be evaluated; under the above assumption:

$$Pr(A_k) = Pr(\bigcup_{i=1}^{n} \{w_{k,i}\}) \bigcup \bigcup_{i=1}^{n-1} \{w_{k,i}w_{k,i+1}\})$$

where $n$ is the requirement vocabulary size. $\{w_{k,i}\}$ were assumed independent events in [3, 14]. However, $\{w_{k,i}\}$ or $\{w_{k,i}w_{k,i+1}\}$, $i = 1, \ldots, n$ independence can be questioned. The document specific weight $\lambda_k \in [0, 1]$ can be introduced to rewrite the equation as:

$$Pr(A_k) = Pr(\bigcup_{i=1}^{n} \{w_{k,i}\}) + \lambda_k Pr(\bigcup_{i=1}^{n-1} \{w_{k,i}w_{k,i+1}\}) \quad (7.1)$$

The problem is decoding the observation $O$, e.g., program item names, into the original high-level document $W_k$. That is, finding $\hat{W}_k$ that maximizes the a-posteriori probability $Pr(W_k \mid O)$ under the assumptions $Pr(W_k \mid O) = Pr(A_k \mid O)$ and
\[ Pr(W_k) = Pr(A_k) \] (i.e., that the document probability matches the abstraction
probability). Applying the Bayes’ rule, the following identity is obtained:

\[
Pr(W_k \mid O) = Pr(A_k \mid O) = \frac{Pr(O \mid A_k)Pr(W_k)}{Pr(O)} \tag{7.2}
\]

The probability \( Pr(W_k) \) is the high-level document \textit{a-priori} probability, since
there is no reason to believe that a high-level text document is more likely than
any other (i.e., there is no \textit{a-priori} information on the document distribution), it
can be safely assumed that documents are equally likely. Furthermore, in the above
equation, \( Pr(O) \) is a constant with respect to \( k \) and could be eliminated. Hence,
decoding is equivalent to find:

\[
\hat{A}_k = \arg \max_{A_k} Pr(O \mid A_k) \tag{7.3}
\]

The high-level document \( W_k \) is transformed by programmers into an abstraction
\( A_k \) down to an observation \( O = \bigcap_{j=1}^{m} \{ o_j \} \), where \( m \) is the code vocabulary size,
corresponding to a low-level artifact (e.g., the identifier collection extracted from a
chunk of source code). Thus,

\[
Pr(O \mid A_k) = Pr(\bigcap_{j=1}^{m} o_j \bigcup_{i} t_{k,i}) \tag{7.4}
\]

Further assuming \( \{ o_1 \} \cap \{ o_2 \} \ldots \cap \{ o_m \} \), conditionally independent from \( A_k \) i.e.,
dependencies between \( \{ o_p \}, \{ o_q \} \) are modeled by the conditioned probability of events
\( \{ o_p \mid A_k \}, \{ o_q \mid A_k \} \). The above equation can be rewritten as:

\[
Pr(O \mid A_k) = \prod_{j=1}^{m} Pr(o_j \mid A_k) \tag{7.5}
\]

In other words, each \( o_j \) depends on the entire abstraction \( A_k \). The conditioned
probability \( Pr(o_j \mid A_k) \) can be simplified as:

\[
Pr(o_j \mid A_k) = \frac{Pr(o_j \cap A_k)}{Pr(A_k)} = \frac{Pr(o_j \cap (\bigcup_{i} t_{k,i}))}{Pr(A_k)}
\]
The event $o_j \cap (\bigcup_i t_{k,i})$ can be rewritten as $\bigcup_i (o_j t_{k,i})$. However, since $t_{k,i}$ were not assumed independent events, when considering single words and couple of adjacent words the following relation holds:

$$Pr(o_j A_k) \simeq Pr(\bigcup_{i=1}^{n} o_j \{w_{k,i}\}) \bigcup_{i=1}^{n-1} o_j \{w_{k,i} w_{k,i+1}\}) =$$

$$Pr(\bigcup_{i=1}^{n} o_j \{w_{k,i}\}) + \lambda_k Pr(\bigcup_{i=1}^{n-1} o_j \{w_{k,i} w_{k,i+1}\})$$

$\lambda_k$ may also be thought of as a scaling factor weighting our belief that a concept is mapped by a bigram. To obtain a tractable form, a $\lambda'_k$ may be chosen so that event dependences are accounted for (e.g., $o_j \{w_{k,i} w_{k,i+1}\}$ versus $o_j \{w_{k,j} w_{k,j+1}\}$):

$$Pr(o_j | A_k) \simeq \sum_{i=1}^{n} Pr(o_j | w_{k,i}) Pr(w_{k,i}) + \lambda'_k \sum_{i=1}^{n-1} Pr(o_j | w_{k,i} w_{k,i+1}) Pr(w_{k,i} w_{k,i+1})$$

(7.6)

By substituting equation (7.6) into equation (7.5) the following expression of $Pr(O | A_k)$ is obtained:

$$Pr(O | A_k) \simeq \prod_{j=1}^{m} \sum_{i=1}^{n} Pr(o_j | w_{k,i}) Pr(w_{k,i}) + \lambda'_k \sum_{i=1}^{n-1} Pr(o_j | w_{k,i} w_{k,i+1}) Pr(w_{k,i} w_{k,i+1})$$

(7.7)

Equations (7.3) and (7.7) are central to the method. They provide a means to effectively recover traceability links based on the a-priori knowledge of a subset of traceability links.

The involved probabilities $Pr(o_j | w_{k,i})$ and $Pr(w_{k,i})$ (as well as $Pr(o_j | w_{k,i} w_{k,i+1})$, $Pr(w_{k,i} w_{k,i+1})$, and $\lambda'_k$) need to be estimated on a labeled training set, i.e., a subset of high-level document and code fragments known to belong to traceability relations. Given the labeled training set for example, the unigram probabilities may be approximated with frequencies:

$$Pr(o_j | w_{k,i}) \simeq \frac{c(o_j w_{k,i})}{c(w_{k,i})} \quad i = 1 \ldots n \quad j = 1 \ldots m$$

(7.8)
where \( c(h) \) is the number of times that the \( h \) word appears in the texts, in the same way \( c(hl) \) is the number of times that the couple \( h,l \) appears (\( h \) in the observation \( O \) and \( l \) in the document \( W_k \)), while \( |W_k| \) is the number of words in \( W_k \) (i.e., \(| | \) gives the set cardinality). A similar approximation allows to estimate the remaining probabilities, while \( \lambda'_k \) are chosen so that they minimize the error over the training set.

### 7.4.1 Model Generation

Probability estimation was based on the word frequency: \( Pr(w_{k,i}) \) is estimated from \( c(w_{k,i}) \), the number of times in which each word appears in a given document. Using the simple word frequency would turn the product \( \prod_{j=1}^{m} Pr(o_j|A_k) \) to zero whenever an observation \( o_j \) was not present in the training set of the document \( W_k \).

This problem, known as the zero-frequency problem [45], can be avoided using different approaches (see [42]). Basically, it has to be decided whether or not the vocabulary is closed. Closed vocabulary means that all the words in the text document collection (set of \( W_k \)) and in all possible observations are known. On the contrary, open dictionary means that a dummy word, accounting for all unforeseen events, is introduced.

While in other fields, such as in spoken dialogue with computers, open vocabulary approaches are more appealing, closed vocabulary does not constitute a limitation in software engineering (i.e., the vocabulary is known once documents are available). Results were obtained with a closed vocabulary where probabilities were smoothed according to the shift-\( \beta \) smoothing techniques. More details and a complete overview on different smoothing techniques and probability estimations can be found in [42].

### 7.4.2 Model Assessment

To obtain an unbiased estimate of the performance of the traceability recovery method, a cross-validation approach was adopted [46] on three different software sys-
tems. Given the requirements to components (e.g., classes of files) traceability map of a software system, components implementing a given requirement were divided into two non-overlapping sets: a training set and a test set. More precisely, the following sets were considered:

- $CS_i$, the set of software ComponentS implementing the $i$-th requirement;
- $TR_i$, the Training Set related to the $i$-th requirement; $TR_i$ components were used to estimate the probabilities required ($TR_i \subseteq CS_i$);
- $TS_i$, the Test Set related to the $i$-th requirement ($TS_i \subset CS_i$ and $TR_i \cap TS_i = \emptyset$), $TS_i$ components were used as queries to evaluate the (7.3); and
- $ES_i$, the set of couples $(TR_{i}, TS_{i})$ related to the $l$-th experiment.

Given a software system such as ALBERGATE, an exhaustive cross-validation on a whole software system was not feasible; there are millions of different and consistent $ES_i$s. For this reason, the experiments aimed to mimic the incremental reconstruction from scratch of traceability links. The probabilities $Pr(a_j|w_{k,i})$, $Pr(a_j|w_{k,i}w_{k,i+1})$ and $\lambda_k$ were thus estimated using $TR_i$ and performance measured over $TS_i$.

Two standard measures for IR, Precision and recall, were used to assess the method results and compare them with previous works [14, 15]. Precision and recall have been defined in Chapter IV. When applying equation (7.3), the most relevant index is precision; a fixed number of models were trained, and thus, a fixed number of document retrieved.

### 7.5 Traceability Problems in Software Systems

Software systems often contain elements that may affect traceability and understandability. We classified these elements, according to the following taxonomy:

1. Partially automatic-generated code;

2. Totally automatic-generated code;
3. COTS and reused code;

4. External Architectures (e.g., middlewares, network infrastructures, etc.); and

5. Design and Implementation-level components.

In the following sub-sections, each factor will be analyzed and relation to traceability issues established.

### 7.5.1 Partially Automatic-Generated Code

These are components and classes, often related to the GUI of the system, containing both automatically generated code (and identifiers) and code written by programmers. For example, identifiers and names of GUI widgets (labels, text fields, buttons, etc.) are automatically generated following precise naming conventions (e.g., TButton1, TButton2, TTextField1, etc.). Naming conventions may be customized; widget identifiers and names changed from a property window of the RAD IDE (Integrated Desktop Environment) to better reflect high-level documents or application domain concepts. However, programmers tend to give significant (and domain consistent) names only to a small fraction of automatically generated component elements; elements they perceive as essential to understand the application while developing the system. The results are that default names dominate (particularly for labels, shapes and any visual component having a constant value and behavior) over programmer specified names.

Similar considerations may be applied to those visual components, encapsulating specific functionalities and dragged into user interfaces. Any kind of visual COTS, such as Microsoft ActiveX or OCX and Borland VCL, belongs to this category. As a second example, consider a component implementing a database management system connection and queries, or components encapsulating network services. The component may have a default generated name (e.g., Table1, Table2, and Query1) or a user-defined name; if the name is not consistent with the related requirements, traceability will be lost. Notice that these components may encapsulate complex
behaviors (update a database table, retrieve a page from the web, print a PDF document, etc.). Thus, the more complex the encapsulated behavior, the more severe the traceability problem is.

To maximize traceability, these default-generated identifiers should be filtered out. For example, two automatically generated windows containing the same number and types of widgets will exhibit very similar or identical identifier frequency distribution, although they are associated to different domain concepts and requirements. Even worse, generated identifiers are likely to dominate over identifiers chosen by programmers, thus reducing the traceability link recovery precision.

### 7.5.2 Totally Automatic-Generated code

These are components and classes automatically generated (e.g., components generating reports, created by report generators) without any need of coding; the programmer intervention, if any, is almost irrelevant (e.g., fill the printer spooler device property of the visual component).

Although these components and classes implement functionalities specified into high-level documents, traceability recovery is infeasible, for the same reasons described in the previous subsection. Suppose the requirement document contains the following sentence:

"The system must allow to print the list of customers from a particular town, specifying names, address..."

Such a requirement is simply implemented by an automatic-generated report, containing only identifiers such as QRBand1, QRBand2, QRLabel1, QRDBText1, QRLabel2, QRDBText2, etc.

To ensure traceability, manual intervention is required, either recording traceability links or assigning consistent names while using the report generator (e.g., the fields QRDBText1 and QRDBText2 should be mapped into Customer_name, Customer_address, etc.).
7.5.3 COTS and Reused Code

Usually, COTS source code is not available; the only traceable elements are the uses of COTS resources (e.g., classes, functions, etc.). If COTS code is available, it is not guaranteed that COTS identifiers reflect the application domain concepts and high-level documents names. This because the component identifiers are mapped to concepts related to the specific solution (e.g., a component for image compression contains identifiers related to Fourier transform, quantization, etc., while the system using it may contain, for example, identifiers related to a geographic information system). Moreover, the component could have been developed in a different organization adopting different naming conventions. As for automatically generated code, manual intervention is required.

Reused components are equivalent to COTS, apart from the availability of source code; the code may have been developed in previous projects possibly related to different application and problem domains, thus consistency between program item names and requirement terms is very unlikely. In other words, reused components and classes should be excluded from traceability recovery process.

7.5.4 External Architectures

This category encompasses middleware (e.g., CORBA) and frameworks (e.g., Zope). External architectures behave similarly, at a different level of granularity, to the partially generated code. System components or system class hierarchies are in relation with other external hierarchies of classes. When an external architecture is integrated, there may be automatically generated source code files or classes (with or without programmer hooked code), classes partially generated, derived from the external hierarchies (with or without overloading) or reused from a framework.

A typical example is the use of an ORB (Object Request Broker) in a distributed software system to guarantee interoperability between application objects and remote objects. A node of the distributed system may contain the following families of classes:
• Classes that implement portion of CORBA (reused classes). As for other reused
code, the traceability link recovery is, in general, not possible;

• Automatic generated classes: these classes are generated by the IDL (Interface
Definition Language) compiler, typically to implement data structures passed
around by distributed objects. Since original variable names (as defined in the
IDL specification files by programmer) are modified and several other variables
are generated, there is no guarantee to recover traceability links; and

• Partially generated classes: these are, for example, the CORBA stubs and
skeletons, containing methods and identifiers with names as defined by pro-
grame in the IDL. These classes can be successfully traced to requirements
specifying information communicated to/from remote systems (implemented
using CORBA).

7.5.5 Design and Implementation-Level Components

These components are not referred into requirements. Therefore there is no guar-
antee to ensure their traceability. Consider, for example, the splash screen or the
about window of a GUI application. Usually, there is no requirement related to these
classes. The motivation is grounded into the object-oriented development process
itself. Requirements are well mapped into classes reflecting the domain-component
view [47, 48] or the conceptual view of the system [39, 49, 50]. At the design level,
the class hierarchy is modified [39, 47, 48, 49, 50], refining domain object and adding
details such as reused classes, human interaction classes, data management classes,
and task management classes. These components and classes cannot be traced di-
rectly to high-level documents, an intermediate step tracing them into design may
be therefore required.

7.6 The Traceability Recovery Process

Figure 7.2 shows the generic process defined to recover traceability links between
high-level documents and source code. The process accounts for the affecting factors
identified in the previous section; it was instantiated in the context of traceability link recovery between functional requirements and the corresponding OO code.

The underlying approach assumes the considerations of Section 7.3, and it further assumes that for each requirement at least one traceability link is available (continuous lines in Figure 7.3). This a-priori information is recovered by a manual analysis, supported by different approaches such as traceability link recovery based on vector spaces [3] integrated with code browsing and visualization. The knowledge of existing traceability links bootstraps the recovery of remaining unknown links (dashed lines in Figure 7.3).

The recovery process can be thought of as composed of three main blocks, each one further decomposed in sub-blocks:
1. Requirements processing;

2. Code processing; and

3. Traceability map recovery by means of the Bayesian classifier.

### 7.6.1 Requirements Processing

Each requirement is processed and the set of its unigrams and bigrams, extracted. Stop words (e.g., articles) are discarded; successively, words have undergone a normalization step of morphological analysis (the *Stemmer* box in Figure 7.2). Plural and singular conjugated verbs, as well as synonyms, are brought back to the radix.

It is worth noting that, in general, these phases cannot be fully automated, and it is likely to remain semi-automatic; it is well known that word semantics may be context sensitive, and currently available natural language tools cannot fully disambiguate contextual semantics, nor deal with complex form representing metaphoric speech (e.g., *...the process should be killed ...*). Finally, the vocabulary is built, weighting each word by its frequency.

### 7.6.2 Code Processing

For each class, a list of identifiers and the associated occurrences is extracted from both interface and implementation files. On the contrary of previous works
[3, 14], comments are not discarded. Extracted program item names are filtered by a stopper. Stop words removal is divided into two sub-phases. In the first sub-phase, the stopper discards the following elements:

- The same stop words removed from requirements;
- Language reserved keywords and language predefined types; and
- Short identifiers, commonly used as loop counters or array indexes (e.g., x, y, j, k).

Since short words may convey relevant application domain information, the phase requires human intervention. The first sub-phase is followed by the removals of automatic-generated identifiers. As highlighted in Section 7.5, they may confuse the traceability recovery. The pruning is performed automatically once the list of classes available in the RAD tool library is known; identifier will have a name generated following well-defined rules (e.g., for the class TLabel generated identifiers will have names such as TLabel1, TLabel2, etc.).

Morphological analysis is subsequently applied to the remaining words. However, the applied analysis is slightly different from those performed on requirements. Plural and singular verb forms are, firstly, normalized; furthermore, method names handling events dispatched by the same object are brought back to a radix equal to the object name. For example, if there is a text field named Address, methods named AddressClick, AddressChange, etc. are brought back to the radix Address. This phase (the box tagged as “Normalizer” in Figure 7.2) can be automatically performed when RAD tools are used; the method names have the object name as prefix, followed by the event name (the list of possible events is known).

Finally, as underlined in Section 7.5, classes, entirely generated, that cannot be traced into high-level documentation, are removed from the system under analysis. This step can be performed automatically once the tools used to develop the system are known.
Reused code and external architectures are processed as untraceable classes, and removed from further considerations. Unfortunately, given the granularity level, the removal process is more human-intensive with respect to automatically generated code. However, file names, comments, documentation (if available) or methods such as those presented in [51] may serve to facilitate the activity.

7.6.3 Traceability Map Recovery

Equations reported in Section 7.4 are applied to the a-priori known links, and probabilities on training material estimated. Finally, the Bayesian classifiers score traceability links with probability. Once the maintainer recovers new links, these are in turn used as new inputs, enriching the classifier training set; the re-trained classifier is then applied to the remaining class-requirement couples.

7.6.4 Tool Support

To automate the traceability recovery process, a number of tools have been developed:

1. A script extracting bigrams and single words, and their occurrences from requirements, plus a scripts that parse the source code (written in Java, C++ or Visual Basic) and extracts the list of couples identifier/occurrence (keywords, language types and language’s symbols are removed);

2. A stemmer that removes stop-words (such as articles, numbers, preposition, certain categories of verbs etc.) from requirements and code. The maintainer is left in charge (by modifying the stop word list) of the final decision whether or not a word should be removed;

3. A morphological analyzer, based on a thesaurus produced by the maintainer from the dictionary of all the possible terms of the requirements/code;

4. A script that, given the names of library classes used by a RAD (or present in a middleware), removes all automatic-generated identifiers from the code; and
5. Finally, a tool that implements the estimation of probabilities $Pr(o_j|w_{k,i})$, $Pr(w_{k,i})$, $Pr(o_j|w_{k,i}w_{k,i+1})$, $Pr(w_{k,i}w_{k,i+1})$ and $\lambda_k'$ with the shift-$\beta$ smoothing and closed vocabulary.

7.7 Case Studies

Feasibility study and method assessment were performed by recovering traceability links between functional requirements and source code of three software systems having different characteristics:

- A Java system, developed following the Boehm waterfall model [52] and implemented without making use of RAD tools;

- A C++ system, developed following a prototyping approach, and making use of RAD tools, COTS and external software architectures; and

- A Visual Basic system, developed following a waterfall model, but implemented making use of the Visual Basic IDE.

7.7.1 Case Study Descriptions

7.7.1.1 ALBERGATE

The software system was developed in Java by a team of final-year students at the University of Verona (Italy). ALBERGATE is a software system designed to implement all the operations required to administrate and manage a small/medium-size hotel (room reservation, bill calculation, etc.). The system has been developed from scratch on the basis of 16 functional requirements specified (as well as all the other system documentation) in Italian language. ALBERGATE exploits a relational database for most of the operations. Therefore, the system size is relatively low (about 20 KLOCs with 95 classes). Non functional requirements, as well as other documentation such as user manuals, UML use cases, architecture and detailed design, were not used in the present study. The recovery process focused on the 60 classes implementing the user interface of the software system.
7.7.1.2 Transient Meter

Transient Meter is a distributed measurement system for power quality monitoring. The system is composed of nodes placed near sites to monitor (e.g., generators, motors, industrial plants), and connected with it by a trigger circuit (that detects the power disturbance) and a digital acquisition board. These nodes detect power transients, process them (performing classification and measurement) and then send them to a central node, that retrieve data from distributed nodes and stores it into a database.

The central node on which the traceability recovery process was applied performs other complex activities such as system startup, configuration monitoring, waveform processing (e.g., Fast Fourier Transform), and visualization.

The system suffered of almost all traceability problems outlined in Section 7.5:

1. The system was developed using a RAD tool (Borland C++ Builder\textsuperscript{TM}), therefore it contains both totally-automatic generated classes (e.g., produced by report generators) and classes containing some identifiers automatically generated (e.g., user interface classes);

2. The system reused existing class hierarchies (e.g., to read and write data from/to .wav files). Moreover, COTS components were used for handling complex signal visualizations, to implement some user interface widgets typical of a measurement instrument (led, switches, seven-segment displays, etc.), and to handle Fourier transforms; and

3. Finally, the communication between the central node and the distributed nodes was implemented by a CORBA architecture.

The SRS document contains 18 requirements, and the source code consists of 63 classes (about 25 KLOCs). The subsystem analyzed (i.e., the central node) has been developed starting from 13 requirements, and it contains 36 classes (about 15 KLOCs). Three classes were automatically generated by the RAD tool, seven are reused classes, four are design-level classes, and three classes belong to the CORBA
architecture. Other classes generated by the IDL compiler to define data structures exchanged between distributed objects (identified by a well-known postfix: _var, _forany) were disregarded.

7.7.1.3 Library Management Software

The third system is a library management software, developed for a university library. The system is composed of two modules:

- A client application, basically a GUI front-end used by dedicated library personnel; and

- A web application, accessible to all users to check the availability of a book.

The traceability recovery experiments were performed on the first application developed from 10 functional requirements. It consists of 35 Visual Basic files (about 16 KLOCs). The intent of the experiment, in this case, was to map requirement onto Visual Basic files. Each file, on its own, may be:

- A frame (having extension .frm) implementing a window of the system, containing the layout of the window, and the event handlers of the widgets (contained into the window) and of the window itself; and

- A module composed of some procedures or functions implementing specific functionalities.

7.7.2 Case Study Results

This section reports the case study results obtained applying the proposed traceability recovery process to the software systems described in Section 7.7. For sake of completeness results, previously published in [3, 36] and [15], are also summarized here, in that they constitute a comparison baseline.

Experiments were also performed, for all the three systems, considering up to three-word terms (i.e., single words, plus bigrams, plus trigrams). However, adding trigrams did not ameliorate performances with respect to bigrams. This may indicate
that, for the three systems analyzed, most of the concepts were retained into terms composed of at most two adjacent words.

Results are reported in tables showing the precision and the recall for different sizes of the training set (i.e., one, two or more known links for each requirement). Precision and recall were computed for different positions in the scoring (i.e., the best one, two, three classes associated to a requirement). The maximum number of classes used as training set for each query depends on the average number of classes associated to each requirement: up to five classes for ALBERGATE, three for Transient Meter and only two for the Library Management software were considered.

To mimic the real world process application, and to obtain an average estimate of the performance under different conditions (i.e., material included/excluded form the training set), experiments were replicated several times (see Section 7.4); each table entry corresponds to the mean computed over 100 random experiments; the number of experiment replications was chosen to guarantee a standard deviation of precision and recall below 3%.

Finally, in order to compare different training set and to evaluate the influence of bigram probability, t-tests (significance level $\alpha = 5\%$) were performed$^1$.

### 7.7.2.1 Choosing the $\lambda'_k$ Factor

As highlighted from equation (7.1) in Section 7.4, performances of the method are influenced by the weight $\lambda'_k$ associated to the multi-word term probability ($\lambda'_k = 0$ for unigram probability). To reduce the computational load, a preliminary phase to assess the effectiveness and influence of $\lambda'_k$ has been performed.

Experiments performed for different values of $\lambda'_k$, varying from zero to one, showed that, on the available case studies, values of $\lambda'_k$ between 0.3 and 1 did not significantly affect results. For $\lambda'_k < 0.3$, results are very close to those for $\lambda'_k = 0$. Moreover, we experienced that, having different $\lambda'_k$ for different documents (i.e., for different values of $k$) did not significantly influence the results. Hence, in this work results

---

$^1$From this point, the terms “significantly affect” or “significant improvement” will be used to state that statistical evidence accordingly to t-test was obtained
were computed assuming $\lambda'_k = 0$ for unigrams and $\lambda'_k = 1$ for unigrams plus bigrams, $\forall$ document $k$.

### 7.7.2.2 ALBERGATE

Table 7.1 reports results obtained with $\lambda'_k = 0$, and applying the Bayesian classifier once:

- Stop words were removed and morphological analysis was applied to requirements; and

- Stop words and normalization were applied to program item names.

The experimental conditions are the same of those giving the best results presented in [14]; results slightly differ, due to fluctuations caused by randomly generating the traceability recovery experiments.

<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Class Training</td>
<td>Precision (%): 33</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 33</td>
<td>45</td>
<td>53</td>
</tr>
<tr>
<td>2 Classes Training</td>
<td>Precision (%): 46</td>
<td>30</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 46</td>
<td>60</td>
<td>68</td>
</tr>
<tr>
<td>3 Classes Training</td>
<td>Precision (%): 51</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 51</td>
<td>65</td>
<td>72</td>
</tr>
<tr>
<td>4 Classes Training</td>
<td>Precision (%): 52</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 52</td>
<td>64</td>
<td>73</td>
</tr>
<tr>
<td>5 Classes Training</td>
<td>Precision (%): 55</td>
<td>34</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 55</td>
<td>67</td>
<td>76</td>
</tr>
</tbody>
</table>

Table 7.1: ALBERGATE $\lambda'_k = 0$ traceability results.

A reference landmark for our results is the simplest way to trace software classes onto requirements: the `grep` UNIX command. The search can be done at least in two ways. In the first approach, each class identifier is used as the string to be searched into the files of requirements; this give rise to about 4800 queries, 94% of which give empty result. The second approach considers the or of the class identifiers, in such a case 94% of the classes were traced onto 10 or more requirements. Even worse, the `grep` approach did not offer any way to rank the retrieved requirements. From a practical point of view, this means that the maintainer has to examine a large number of candidates with the same priority.
The subsequent step (see Table 7.2) considers $\lambda_k' = 1$, corresponding to the situation where both single words and bigrams are modeled.

A comparison between results obtained here and those published in [3] and [14] is very encouraging, and supports the newly proposed equations. There is an increase in precision and recall: Table 7.2 results outperform Table 7.1, due to the bigrams contribution. Comparison with [36] and [3] is somehow more difficult, in that the approach is almost completely different; [3, 36] use a unique dictionary: program item names are forced to appear into requirement. Moreover, the equation proposed in [3] could not be re-estimated once new training material is available. Table 7.2 results are significantly better than those in [3, 36], once sufficient training material is available (i.e., two or more links).

Current results improve recall and precision: although using one-class training set the performance does not significantly improve, results significantly ameliorate with training sets of two or three classes for the Best 1 and Best 2 ranking scores. However, with respect to the $\lambda_k' = 0$ model, when increasing the training set performances may deteriorate due to the following factors:

- As explained in the introduction, when considering bigrams, the number of possible terms is upped bounded by twice the number of words in the text documents (i.e., the denominator of the equation (7.9) doubles). In particular, when a single class is available for training, bigram probabilities did not ameliorate the results; and

- Adding new traceability links may introduce useless information that deteriorates the signal to noise ratio. In other words, when four or five classes are used to train the system, most of the requirements are already mapped to the respective implementing classes. Therefore, the test set includes only a few classes and, furthermore, those classes tend to have terms in common thus decreasing the method discriminant power.
<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Class</td>
<td>Precision (%): 33</td>
<td>22</td>
<td>20</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 33</td>
<td>45</td>
<td>57</td>
</tr>
<tr>
<td>2 Classes</td>
<td>Precision (%): 54</td>
<td>32</td>
<td>23</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 54</td>
<td>64</td>
<td>69</td>
</tr>
<tr>
<td>3 Classes</td>
<td>Precision (%): 57</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 57</td>
<td>66</td>
<td>72</td>
</tr>
<tr>
<td>4 Classes</td>
<td>Precision (%): 54</td>
<td>33</td>
<td>24</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 54</td>
<td>67</td>
<td>72</td>
</tr>
<tr>
<td>5 Classes</td>
<td>Precision (%): 53</td>
<td>32</td>
<td>23</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 53</td>
<td>64</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 7.2: ALBERGATE single words plus bigrams traceability recovery results.

7.7.2.3 Transient Meter

Consistently with the ALBERGATE case study, a baseline was established applying the process proposed in [14] (i.e., $\lambda_k = 0$), and results calculated:

1. After the morphological analysis (including stop words removals);

2. After removal of automatic-generated identifiers; and

3. After excluding non-traceable classes.

Table 7.3 results were obtained with an experimental setup corresponding to the first item; at a first glance, it immediately appears that traceability recovery is quite poor, when compared to results of Table 7.1 and Table 7.2.

<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Class</td>
<td>Precision (%): 7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 7</td>
<td>16</td>
<td>26</td>
</tr>
<tr>
<td>2 Classes</td>
<td>Precision (%): 4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 4</td>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>3 Classes</td>
<td>Precision (%): 6</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 6</td>
<td>16</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 7.3: Transient Meter $\lambda_k = 0$ baseline results.

Differences were explained in terms of the adopted software development approaches. Transient Meter was developed with RAD IDE, COTS, communication middleware, reused code and the other traceability affecting factors described in Section 7.5. On the contrary, ALBERGATE was entirely coded from scratch without reusing components, but integrating a relational database.
Data reported in Table 7.3 are puzzling, they seem to contradict the learning effect experienced on the ALBERGATE case study: adding information decreases traceability. As new training material was added, and new links were recovered from the system, precision and recall resulted as worsened.

The counterintuitive phenomenon happens when the training material added has few (or no) common words (i.e., identifiers) with the pre-existent training set (i.e., classes previously associated with the given requirement). The situation violates the assumption that knowledge is processed consistently; the new training material (i.e., program item names) adds noise rather than useful information. Probability distributions are flattened, causing classes of the test set to be easily associated with a wrong requirement.

Although the steps described in Figure 7.2 to process the source code do not always ameliorate precision, it has been demonstrated [15] that each step is an essential component of the traceability recovery process, when dealing with COTS, middleware, RAD IDE, etc. As underlined, automatic-generated identifiers need to be pruned to avoid flattening probability distribution with spurious information. Results obtained after this step are shown in Table 7.4.

<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Class Training</td>
<td>Precision (%): 5</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 5</td>
<td>18</td>
<td>31</td>
</tr>
<tr>
<td>2 Classes Training</td>
<td>Precision (%): 1</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 1</td>
<td>15</td>
<td>32</td>
</tr>
<tr>
<td>3 Classes Training</td>
<td>Precision (%): 2</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Recall (%): 2</td>
<td>18</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 7.4: Transient Meter: $\lambda'_k = 0$ results after removal of automatic-generated identifiers.

The next step, the normalization phase, helps to associate classes whose identifiers (attributes, methods or comments) share common radices. However, in presence of automatically generated code and COTS, the normalization tends to add noise, more precisely confounding links, thus automatically generated code and COTS need to be removed.
**Transient Meter** contains classes that cannot be effectively traced into requirements; these classes consist of one splash screen, a window to display aggregated data from the database, and a class for report printing. It was also discovered that, in **Transient Meter**, reused classes have no way to be traced into requirements. Thus, seven classes (two for handling wave files, five for signal processing), all mapped to a single requirement, were excluded prior to compute new results. Moreover, classes belonging to the CORBA architecture (three classes), except the stub, were removed. Finally, some low-level design classes, which cannot be directly mapped to requirements, were identified. A total of four classes, three handling data structures and one implementing an adapter to reused classes, were removed, thus the final number of classes traced into requirements was 19 (out of the 36 initial).

By applying the process of Figure 7.2, encompassing the described activities accounting for the traceability affecting factors (see Section 7.5), Table 7.5 results were obtained, showing that the system significantly learns as the training set increases.

<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Class Training</td>
<td>Precision (%):</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>Recall (%):</td>
<td>26</td>
<td>48</td>
<td>60</td>
</tr>
<tr>
<td>2 Classes Training</td>
<td>Precision (%):</td>
<td>60</td>
<td>38</td>
</tr>
<tr>
<td>Recall (%):</td>
<td>60</td>
<td>76</td>
<td>83</td>
</tr>
<tr>
<td>3 Classes Training</td>
<td>Precision (%):</td>
<td>71</td>
<td>46</td>
</tr>
<tr>
<td>Recall (%):</td>
<td>71</td>
<td>92</td>
<td>93</td>
</tr>
</tbody>
</table>

Table 7.5: **Transient Meter**: \( \lambda_k^1 = 0 \) final results (after removal of non-traceable classes).

Finally, the experiments corresponding to \( \lambda_k^1 = 1 \) were ran on the same material used to obtain Table 7.5. In other words, traceability recovery was computed based on the equation (7.1), where bigram terms play a fundamental role; results shown in Table 7.6 confirm the ability of the approach to significantly learn.

Moreover, with respect to the unigrams, there is a significant performance improvement for the top ranking score (Best 1). This means that some classes ranked in second or third position using unigrams, obtained the Best 1 ranking score thanks to bigram contribution. On the other hand, bigrams did not help so much in correctly ranking more classes than unigrams among the top three scores.
Table 7.6: Transient Meter: single words plus bigrams traceability results.

### 7.7.2.4 Library Management Software

The traceability recovery process followed for this system was the same adopted for *Transient Meter*. In particular, after removing non-traceable files (3 out of 25 files, implementing utility functions, were removed).

Results are shown in Table 7.7, 7.8 and 7.9. The performance improvement after the different steps basically confirms results obtained for the two previous analyzed systems.

<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 File</td>
<td>Precision (%): 35</td>
<td>26</td>
<td>21</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 35</td>
<td>47</td>
<td>64</td>
</tr>
<tr>
<td>2 Files</td>
<td>Precision (%): 33</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 33</td>
<td>52</td>
<td>62</td>
</tr>
</tbody>
</table>

Table 7.7: Library Management: $\lambda'_k = 0$ results after morphological analysis.

<table>
<thead>
<tr>
<th>Score:</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 File</td>
<td>Precision (%): 33</td>
<td>27</td>
<td>23</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 33</td>
<td>55</td>
<td>68</td>
</tr>
<tr>
<td>2 Files</td>
<td>Precision (%): 34</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%): 34</td>
<td>64</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 7.8: Library Management: $\lambda'_k = 0$ results after removing automatic-generated identifiers.

Finally, Table 7.10 reports results obtained considering bigrams. This system is an example where considering bigrams did not significantly help (on the contrary, performances for the top score tends to decrease).
Table 7.9: Library Management: \( \lambda_k^t = 0 \) results after removing non-traceable files.

<table>
<thead>
<tr>
<th>Score</th>
<th>Best 1</th>
<th>Best 2</th>
<th>Best 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 File Training</td>
<td>Precision (%):</td>
<td>38</td>
<td>29</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%):</td>
<td>38</td>
<td>58</td>
</tr>
<tr>
<td>2 Files Training</td>
<td>Precision (%):</td>
<td>34</td>
<td>37</td>
</tr>
<tr>
<td>Training</td>
<td>Recall (%):</td>
<td>34</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 7.10: Library Management: single words plus bigrams traceability results.

7.8 Lessons Learned

It is not unlikely that industrial software contains COTS, reused code, communication middleware and, more generally, components that are difficult or even impossible to trace into requirements. The detailed analysis of our case studies revealed that several of such components were present. As for automatically generated GUI, there is no way to automatically trace those low-level artifacts into high-level documentation. Investigation should be made to trace these artifacts to lower-level documents (e.g., design documentation).

When using a RAD IDE environment, reused code, external architectures/ middleware, programmers tend to assign meaningful names only to a fraction of the identifiers. As a consequence, non-domain specific names may dominate over domain related names, thus confusing the traceability recovery process. Untraceable elements should be removed from the analysis; in the case of automatically generated code, a heuristic was adopted to discard classes that, after pruning automatic generated identifiers, exhibited a list of identifiers empty or below a fixed threshold.

While performing the traceability recovery process, we discovered that comments are a valuable source of information, thus, on the contrary of some previous works [3, 14], comments were exploited to recover traceability links. Clearly, this required a coding standard to help associating comments with classes and methods.

In agreement with other works [3, 14], text normalization is fundamental. How-
ever, as highlighted in [15], to obtain benefits from normalization, it is mandatory a strict compliance with the proposed process. Moreover, particular attention should be paid to normalize identifiers (e.g., bringing back object names and event handlers to the same radix).

As in [3, 14] and [15] it has been observed that, in general, once the process is applied, enriching the training set increases the precision/recall of subsequent steps. This fact is very relevant, since other methods, such as those proposed in [3], did not benefit from adding further information.

Statistical tests showed that bigram behaved differently for the three systems analyzed. The general lesson that can be learned is, that, the influence of bigrams depends on the way requirement terms are mapped to code. If, in most cases, single words from requirements are mapped to code variables (e.g., Library Management, the smallest and simplest system), then bigrams confuse the Bayesian classifier. On the other hand, when more adjacent words from requirements are generally associated to variables (e.g., Transient Meter), bigram-based classifier tends to outperform the unigram-based classifier.

Moreover, it has been experienced that the maximum number of adjacent words to consider is limited. In particular, experiments performed on the available systems revealed that training with more than two adjacent words did not improve accuracy.

Attention should also be paid when increasing the training set, since the amount of multiple-word terms increase not only the useful information, but also the noise. Finally, as highlighted in Section 7.7.2.1, a conservative choice of $\lambda'_{k} = 1$ ensures, in the case studies presented, good performances. the choice of the best $\lambda'_{k}$, i.e., the best weighting factor for the multiple word term probability, is relevant to improve accuracy.

### 7.9 Conclusions and Future Work

A new method to recover traceability links between high level documentation, such as functional requirements or use cases, and low level artifacts i.e., detailed design or source code, and the method equations, has been presented.
Central to the method is the assumption that programmers tend to process application-domain knowledge in a consistent way, applying a set of unknown rules, when writing code, more precisely when choosing program item names. Thus, program item names of different code chunks, related to the same high level documents and/or concepts, are likely to be the same or very similar. The method infers in a probabilistic form program behavior, i.e., the rules adopted by the programmers choosing identifiers, those rules are implicitly represented by the joint probability distributions estimated on training sets.

The method was applied to three different software systems; code regions (i.e., classes and files) were traced into the functional requirements; accuracy was evaluated comparing results with the traceability matrices compiled by the system developers. The three case studies can be thought of as representative of different development approaches, languages and tools. Traceability links were recovered in systems developed with RAD IDE, code generators, incorporating databases, middleware and reused code.

Noticeably, on most cases, as the training set increase, the method performance improved; in other words, on the available data the method learns (i.e., the joint probability distribution seems to effectively capture the consistency rules applied when creating identifiers). The task of recovering traceability links may be eased by enforcing appropriate coding standards.

Future works will be devoted to apply the process to different case studies, and to recover a complete traceability map through the documents produced at different phases of the software life-cycle (i.e., use cases, design documents, testing documents, etc.). A tool, able to use traceability recovery information to support the programmers by highlighting how he/she is consistently using identifiers, is also under development.
Bibliography


The reusable software library. In W. Tracz, editor, Software Reuse: Emerging 


[33] J.I. Maletic and A. Marcus. Supporting program comprehension using semantic 
and structural information. In Proc. of 23rd International Conference on 

[34] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to doc-
umentation links in oo systems. Proc. of the Working Conference on Reverse 

models for recovering traceability links between code and documentation. In Proceedings of IEEE International Conference on Software Maintenance, San 

object-oriented code into functional requirements. In Proceedings of the 8th 
International Workshop on Program Comprehension, pages 227–230. Limerick, 
Ireland, June 2000.

[37] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code trace-
ability links using latent semantic indexing. In Proceedings of the International


CHAPTER VIII

Conclusions

About 15 years have passed from the dawn of reverse engineering. Many problems still remain to be addressed, and better solutions to already solved problems are still welcome. Some examples may be a better software visualization, a clustering more focused to what are the maintainer’s objectives, a more precise recovery of information such as the presence of design patterns, traceability links, etc.

On the other hand, new technologies emerge: the user interface of software systems is essentially moving to the web; many applications are ported on small-limited resource devices (such as palmtops or multimedia cellphones); a web-service oriented approach is strongly influencing component-based software engineering. Old, solid solutions are being be applied to new categories of software systems (e.g., applying reverse engineering techniques to web applications, understanding a distributed system, etc.) and for different purposes (e.g., porting a big system on limited-resource devices).

At the same time, new approaches or technologies pose new challenges. For example, the spreading of agile methodologies could have as a consequence, during maintenance tasks, the need for a massive reverse engineering support to fill the lack of documentation produced (because of the limited budget or the strict time-to-market). Finally, as software evolves, its entropy increases: the documentation is not aligned anymore, pieces of code are cloned, libraries grow, the organization of source files into directories becomes chaotic, new configurations may be the cause of possible, unexpected, inconsistencies and can therefore produce of serious problems.
The *Evolution Doctor* proposes possible approaches to monitor the evolution of a software system, forecasting how the system will grow, what will be the influence of clones in the next releases, analyzing multi-configuration code and investigating the quality of OO code in terms of design patterns. Then, it proposes approaches to redocument the system, for example recovering traceability links. Finally, it helps in the revitalization of the system, improving the directory organization of source files and, above all, performing several kind of intervention to improve the organization of software modules into libraries, especially when there is the need of porting the software on limited-resource devices. Case studies presented (most of which large, well-known open source software systems) demonstrated the effectiveness of the proposed approaches.

As explained across all the chapters, there will be a lot of space for further investigations on the different problems encountered and improvements for the proposed approaches.
APPENDICES
APPENDIX A

OCL Based AST Navigator

A.1 Introduction

In the previous chapters several approaches, devoted to analyze, comprehend and to reengineer software systems, have been described. Most of these approaches rely from the analysis of source code, to compute metrics, to extract identifiers, to recover structural relationships, etc. Although the proposed approaches cover a wide variety of aspects, often maintainers need to extract customized properties from source code, to retrieve portions of AST satisfying a given property.

Integrated development environments (e.g., Microsoft Visual C++\textsuperscript{TM}, Borland C++ Builder\textsuperscript{TM}/JBuilder\textsuperscript{TM}, etc.) provide a useful means supporting developer basic operations. However, these tools have limited customization and programming capability and thus they play a little or no role when concepts needs to be located in large software system.

Very promising technologies, such as program slicing [1, 2, 3] and impact analysis [4, 5, 6, 7, 8], have been sometimes integrated into industrial environments, such as GrammaTech CodeSurfer, Semantic Designs Source Code Browser, Codecrawler, Rigi and many others. These tools have very powerful and well-known capabilities, though they have been often designed to solve particular categories of problems. For example, CodeSurfer [9] is particularly indicated for point-to analysis and slicing, Rigi [10, 11, 12] was conceived as a visual tool to help program comprehension and reverse engineering, Codecrawler [13] is a language independent reverse engineering
tool integrated with metrics computation and large software system visualization capabilities, *Semantic Designs Source Code Browser* [14] helps program comprehension by allowing navigation of source code extracted documentation and hyperlinked Java code. The drawback is that these tools may be difficult to integrate with traditional programming environments/languages to build custom tools.

Several categories of languages/toolkits for source code analysis and transformation have been developed during the last decade. Among all, those providing a powerful language oriented to program comprehension and transformation are the *Design Maintenance Systems* (DMS) produced by Semantic Designs Inc. [15, 16], the *TXL programming language* produced by TXL Software Research Inc. [17, 18], *Refine* [19] produced by Reasoning Systems Inc., and *Fermat* by Martin Ward [20, 21]. These tools have powerful analysis capabilities, if compared to standard development environments; e.g., they provide pattern-matching languages and a way to query and transform the AST produced by a parser. By providing mechanisms to query and transform an AST, they can be used to fulfill several analysis and comprehension tasks, as well as to carry out maintenance and source code transformation tasks. However, such tools require trained and skilled people, in that they define proprietary languages.

A well-known language should therefore be used to navigate AST and to perform query on it. The OCL [22] is a good, potential candidate language to browse, navigate, and query, via an underlying programming language object model, ASTs [23].

OCL is a formal language easy to read and to write, developed to specify constraints and more general expressions; OCL has been used in the UML semantics documents and could be considered part of the UML users’ background.

By adopting OCL several other advantages are readily available. First and foremost, the analysis paradigm changes: the focus is no longer an AST or parse tree paradigm, rather an OO paradigm. The idea is not new and to same extent was already introduced by tools such as *JavaCC* [24] and books such as [25]. *JavaCC* allows to generate an AST based on a class hierarchy, and it provides also a mechanism, based on the *Visitor* design pattern [26] to navigate and manipulate it. In [25]
a clean and elegant OO approach to develop compilers is presented even if the author position is clear and, due to porting issues, more inclined to non-OO style in writing code. The second advantage is that UML and OCL are *de-facto* standards, thus the learning curve can be reduced. Third, OCL is powerful enough to express in a concise and elegant way very complex conditions. Finally, by changing the underlying language model, OCL based applications can be easily ported from one language to another (e.g., from Java to C).

Section A.2 presents the language AST object model; a brief overview of OCL basic notions is reported in Section A.3. Section A.4 discusses, presenting some examples, how OCL can be used as an AST navigation and query language. Section A.5 presents the experience in developing the OCL interpreter; finally, Section A.6 gives concluding remarks and outlines directions for future work.

### A.2 The AST Object Model

![AST object model example: method declarator](image)

To navigate and query an AST (extracted from a chunk of source code) using OCL, an object model (representing the underlying source code programming language) has to be preliminary defined. Rules to map a programming language grammar to a class hierarchy have been defined in [25], and can be summarized as follows:
1. A tree is described by one or more abstract classes, corresponding to non-terminal symbols of the grammar;

2. Each abstract class is extended by one or more subclasses, one for each grammar rule in which the abstract class appears on the left side; and

3. For each nontrivial symbol in the right side of a rule, there will be one field (i.e., attribute, association or aggregation) in the corresponding class.

The proposed language object model is slightly more complex, to ease the navigability of the AST itself. In particular, it considers the way in which grammar rules are represented in JavaCC:

- If there are more grammar rules with the same left-hand side, they may be expressed using the option \"|\" operator, e.g., \texttt{cond\_statement = if\_statement | for\_statement | while\_statement}; this, again, is mapped, in the object model, in a class hierarchy where the left-hand side is the superclass and the options on the right-hand side the subclasses; and

- Sequences and options are expressed using the \+ (one or more), \* (zero or more) and \[\] (optional) operators, e.g., \texttt{argument\_list = expression \("," expression)\*}. Such sequences and options will be translated, in the object model, in associations with target multiplicities 1..*, 0..* and 0..1, respectively.

Classes representing the grammar are derived from a superclass \texttt{Node}; \texttt{Node} is the supertype for all nodes in the language object model and it factors out the elements useful to query and navigate the AST. The following associations and attributes are defined via the superclass \texttt{Node}:

- Each node is associated to (all) its children. This will allow to consider the children of a node as an OCL Sequence (as described in Section A.3, something similar to an array), and to iterate through it;
• For similar reasons, each node is associated with (all) its descendants; as shown
in Section A.4, this kind of links are particularly useful to locate, inside an AST,
all the nodes satisfying a specified property;

• Each node is directly associated with (all) its ancestors. This is useful, for ex-
ample, to perform backtracking analysis, such as retrieving (see Section A.4.1)
the conditional statements influencing the execution of a particular statement;

• Each node is directly associated with (all) its descendant tokens (terminal
nodes);

• Each node has two attributes, begin_line and end_line, indicating at which
line of the file the node begins and ends. As shown in Section A.4, this can be
used, for example, to compute the LOCs metric; and

• Finally, each node is associated with its image (i.e., the string associated to it).

Clearly, having associations to all the descendants and ancestors does not neces-
sarily imply the actual existence of such links. In the model implementation, each
node is only associated to its parent and its children. The Sequence of ancestors and
descendants is built on the fly, when needed, using appropriate visitors.

Figure A.2 shows an excerpt of the object model corresponding to the Java
method declaration as represented in the JavaCC grammar. Several JavaCC gram-
mars are available at the JavaCC grammar repository [24]. The grammar was mod-
ified superimposing via the grammar superclass the navigation structure: as shown,
all classes are subtypes of the abstract class Node. A MethodDeclaration has asso-
ciations with its children, i.e., ResultType, NameList, MethodDeclarator and
MethodBlock. A MethodBlock contains zero or more objects of type BlockStatement.
A BlockStatement, in turn, may be a local variable declarator, an unmodified class
declarator or a statement.
A.3 OCL Overview

OCL is a formal language used to express constraints on a UML model, i.e., conditions that must hold on the system being modeled. OCL was created to overcome the limitation of UML to provide all the relevant aspects of the specification. In particular, UML does not provide a formal mechanism to express constraints on diagrams and to specify pre and post conditions for class methods, class invariants, nor guard conditions on statecharts and interaction diagrams.

OCL is a pure expression language, therefore OCL expressions are guaranteed to be without side effects, i.e., any OCL expression simply returns a value, and it cannot change anything in the model. This section will highlight the OCL characteristics that are the most relevant to define an AST navigation language. Further details can be found in [22].

In order to express constraints, OCL allows navigability among classes in a UML meta-model. OCL supports two navigation forms. The first is indicated by “." (as for accessing to a class attribute or method or to apply a feature to a single object), while the fact that a feature is applied on a Collection (see below) is indicated by the symbol “− >”. When navigating through an object model, class names are indicated in lower cases.

Given the AST object model shown in Figure A.2, from an object \texttt{m} of type \texttt{MethodDeclaration} one can access to the line where its \texttt{MethodBlock} begins, simply writing \texttt{m.methodDeclaration.begin_line}. As shown in Section A.4, this formalism is particularly useful for navigating from an AST node to its children (i.e., from a class to its associated classes).

A useful feature is the ability to verify if an object is an instance of a given type, or if it is of one of the supertypes of the given type. This can be done by applying to an object, respectively, the features \texttt{oclIsTypeOf} and \texttt{oclIsKindOf}. Thus, given a node \texttt{v} of type \texttt{Statement}, the expression \texttt{m.oclIsTypeOf(Statement)} returns \texttt{true}, as well as the expression \texttt{m.oclIsKindOf(BlockStatement)}.

Another interesting feature is the ability to access all the instances of a class, e.g.,
MethodDeclaration.allInstances. In this case, the class name has to be written using the same case of the class diagram.

In addition to classes specified in any meta-model, OCL defines:

- Basic types: Integer, Real, Boolean, String, each one provided with a set of the most important features (e.g., basic String manipulation features are available); and

- Collection types: the result of navigation is, in general, a Collection. In particular, the result of a single navigation is a Set, the result of a combined navigation is a Bag, and the result of the navigation over an {ordered} association is a Sequence.

Casting between different Collection types is supported via the oclAsType operator; Collection types are provided with a powerful set of operator enabling the creation of new Collections from existing ones.

This is a key factor and the main motivation underlying the choice of OCL to express the navigation and the query of an AST. The idea is not new, for example, a powerful set of operators to handle collections and sets was present in the Reasoning Software Refinery language [19]. The following is a brief summary of the OCL available constructs operating over collections:

- Extract from a collection all the items (not) verifying a given property - select (reject); perform a projection (collect);

- Universal quantifiers: forall, exists;

- Iterate through all the items of a collection verifying a given conditions, possibly, updating the value of an accumulator variable (returned at the end of the iteration);

- Basic set operators: set union, intersection, difference, set cardinality as well as operator to test if an item belongs to a collection, etc.; and
• Sequence operators: accessing to the first, last, or \(i\)-th item, extracting a sub-sequence, etc.

### A.4 OCL for Navigating ASTs

```
extractClasses(cu: CompilationUnit): Sequence(UnmodifiedClassDeclaration)
post: result = u.descendants->select(oclIsTypeOf(UnmodifiedClassDeclaration))
```

```
extractMethods(cu: CompilationUnit): Sequence(ClassBodyDeclaration)
post: result = u.descendants->select(oclIsTypeOf(MethodDeclaration))
    or oclIsTypeOf(ConstructorDeclaration))
```

```
Conditioning(s: Statement): Sequence(Statement)
post: result = s.ancestors->select(oclIsTypeOf(ForStatement))
    or oclIsTypeOf(WhileStatement)
    or oclIsTypeOf(DoStatement)
    or oclIsTypeOf(SwitchStatement)
    or oclIsTypeOf(IfStatement)
)
```

Figure A.2: Extracting nodes from a tree.

The aim of this Section is to demonstrate with some examples how OCL can be used to fulfill the typical needs of an AST navigation language supporting program comprehension. The examples will be referred to the Java programming language, and the object model corresponds to the Java grammar available at [24].

In particular, we will discuss how to implement the following features:

• Returning some subtrees of an AST: e.g., all the classes declared in a Java file or all the method declared inside a Java class;

• Matching subtrees;

• Searching for all AST nodes having a given property and, in particular, computing some metrics. It will be shown how some of the metrics used to perform a metric-based clone detection [27, 28, 29] can be computed.

AST navigation languages, as mentioned in the introduction, are very often integrated with construct to help AST transformation. However, OCL is a pure expression language, and thus it cannot be used to describe transformations.
Operations performed with OCL are expressed, in this Section, using the standard notation adopted to describe post conditions:

```plaintext
MethodName ""("" [ paramName '::' paramType
    ("",:) paramName '::' paramType)* "")"
''::' ReturnType
```

post: oclExpression

### A.4.1 Extracting Nodes Having a Given Property

The first operation to perform, prior to execute any program comprehension task, is to extract the AST subtrees subject of our analysis. Normally, what we obtain from the parser is an AST for each source file parsed or, if our parser is able to perform a multi-file parsing, an AST forest. Let us now suppose, for sake of simplicity, that our parser works, as in the former case, on a single file, then we may be interested to extract, from a source file:

- The Sequence of all declared classes;
- For each class, the Sequence of its methods; and
- For each class, the Sequence of all its attributes.

The first task can be performed, given a Compilation Unit cu (i.e., the root of a source file AST), by the function `extractClasses` shown in Figure A.4. The `select` feature is applied to all the descendant nodes of cu (i.e., to all nodes of the source file AST). It selects only the nodes of type `UnmodifiedClassDeclaration` (the condition is expressed using the `oclIsTypeOf` feature), and then returns them as a Sequence.

Similarly, the function `extractMethods` extracts all methods declared inside a class `c`.

AST nodes corresponding to attributes and other fields can be similarly matched.

Finally, another interesting example: given a statement `s`, one may be interested to retrieve the Sequence of conditional statements influencing its execution. This can be done using the function `Conditioning` that, as shown in Figure A.4, takes advantage of the association a node has with all its ancestors.
A.4.2 Matching Subtrees

The second feature that an AST navigation language should implement is the ability to perform some forms of tree/subtree matching. In particular, two kinds of operators have been defined:

1. An operator to determine if two trees exactly match, i.e., if the root nodes of the two trees contain the same number of children (both terminals and non-terminals) and all children match; and

2. An operator to determine if two trees structurally match, i.e., if the root nodes of the two trees contain the same number of children (both terminal and non-terminals) and all non-terminal children match.

The first operator was associated with the "=" OCL operator, that returns a Boolean value indicating if two objects match. For the second operator, OCL has been extended with a feature structMatch(target: oclAny):Boolean that applies on any OCL object, i.e., on only AST node in our case.

To better understand the details, let us consider two examples. Firstly, given an Expression e, we need the list of all conditional statements of a method m where the condition is exactly e, i.e., the condition must be expressed on the same variables
with the same constants. Thus, we may write the function \texttt{matchExpression} shown in Figure A.4.1. As shown, the function iterates on all the descendants nodes of the Method Block of \texttt{m}. The feature \texttt{select} provides to return only the statements \texttt{s} satisfying the enclosed condition, i.e., \texttt{s} must be a conditional statement, and its conditional expression must \textit{exactly} correspond to \texttt{e}.

\begin{verbatim}
Parameters(m: MethodDeclarator): Integer
post: result=m.namelist.formalparameters.formalparameter->size()

LOCs(m: MethodDeclarator): Integer
post: result=m.methodblock.end_line-m.methodblock.begin_line

Statements(m: MyMethodDeclarator): Integer
post: result=m.methodblock.descendants->select(oclIsKindOf(BlockStatement))->size()

ReturnStatements(m: MethodDeclarator): Integer
post: result=m.methodblock.descendants->select(oclIsTypeOf(ReturnStatement))->size()

Cyclomatic(m: MyMethodDeclarator): Integer
post: result=m.methodblock.descendants->select(oclIsTypeOf(ForStatement)
    or oclIsTypeOf(WhileStatement)
    or oclIsTypeOf(DoStatement)
    or oclIsTypeOf(IfStatement)
    or oclIsTypeOf(SwitchLabel))->size()+1

Locals(b:MethodBlock): Integer
post: result=b.blockstatement->iterate(n:Node; r:Integer |
    if oclIsTypeOf(LocalVariableDeclaration)
        then r+n.variabledeclarator->size()
    endif)
\end{verbatim}

Figure A.4: Computing metrics.

The second example aims to find \textit{cloned} methods contained in a file. Literature reports several methods to detect \textit{clones} (see Section 2.3.2). For illustrating the \textit{structural} matching feature, we will refer to the clone detection method based on matching subtrees [30]. Given the \texttt{c} the \textit{CompilationUnit}, i.e., the root node of the file parse tree, we may define the function \textit{detectClones} shown in Figure A.4.1.

The OCL feature \texttt{iterate} has been used. This feature iterates on all the items of the Sequence composed of all \texttt{c} descendants and, for each node \texttt{n} a test is carried out to verify whether or not a subtree rooted in \texttt{n} matches any other subtree (rooted in \texttt{n1}). The functionality requires, as shown in Figure A.4.1, the \textit{exists} feature.
The user-defined \textit{structMatch} feature checks if two nodes have the same number of children, and all their children having equal position also satisfy the \textit{structMatch} feature.

\subsection{A.4.3 Computing Metrics}

Given a method \texttt{m} from a Java class, let us suppose we want to compute some metrics on it such as metrics related to size (LOC, number of statements), complexity (e.g., McCabe complexity), coupling (e.g., method passed parameters, local or global variables).

To compute the number of method's passed parameters, the function \texttt{Parameters} shown in Figure A.4.2 can be used. The function shows a classical example of navigation through the UML model of an AST: a \texttt{NameList} is part of a \texttt{MethodDeclarator}, and \texttt{FormalParameters} on its turn, is part of a \texttt{NameList}. The value is computed as the size of the node \texttt{Sequence} by applying the feature \texttt{size()}.

The LOCs may be computed as the difference between the beginning and the end of a method as described in the function \texttt{LOCs}. It takes a \texttt{MethodDeclarator} node as parameter, and then returns the difference between its bounding block \texttt{end_line} and \texttt{begin_line}.

The function \texttt{Statements} (see Figure A.4.2) computes the number of statements composing a method. The feature \texttt{select} takes all the descendants nodes of the method block (i.e., all nodes of the method block AST) and returns the \texttt{Sequence} of nodes satisfying the specified condition: nodes derived from the \texttt{BlockStatement} (i.e., Statement, variable declarator, or unmodified class declarator). Then, as in the previous case, the feature \texttt{size} returns the \texttt{Sequence} size. Much in the same way, other elementary metrics such as the number of return statements (\texttt{ReturnStatements} function) or the number of methods calls can be computed.

The cyclomatic complexity (see function \texttt{Cyclomatic} in Figure A.4.2) can be computed counting all the decision points in a block and then adding 1 [31].

Slightly more complex is the task of counting the number of variables declared in a block \texttt{b} (i.e., all the local variables of that block). This means that, for each block
statement of type \texttt{VariableDeclarator}, we need to count the number of declared variables. Such operation can be expressed in OCL as described in Figure A.4.2 by the function \texttt{Locals(b:MethodBlock)}. The function iterates on all items of the \texttt{SequenceBlockStatement} and, for each node \texttt{n} of type \texttt{LocalVariableDeclaration}, it counts all the items of the \texttt{SequenceVariableDeclarator}, incrementing the Integer accumulator \texttt{r} (to be returned at the end of the iteration).

\subsection*{A.4.4 Discussion}

Although not explicitly designed to navigate ASTs, OCL allows easily expressing navigation and query constructs. This is not surprising, in that the goal of OCL is to express pre and post conditions, as well as guard conditions on UML diagrams. Once mapped into an object model, the grammar of a programming language corresponds to an UML object model, and thus OCL can be used to state properties and conditions over the grammar representation itself.

As shown, it has been possible to express in OCL all the features described at the beginning of this Section. Such features represent, in the opinion of the authors, the basic querying tasks to be performed on an AST. Moreover, the OCL expressions are in general, not very complex, easy to be understood and maintained. This is due to the powerful language capability of:

1. Navigating an object model; and

2. Executing query operations on collections and sets (also present is some other proprietary languages, such as \texttt{Refine}).

One (intentional) limitation of OCL is that it is not a programming language, therefore it cannot express program logic or flow control. This does not prevent to express possible queries useful for program comprehension purposes. However, to perform composite tasks or tasks with side effects (e.g., clone detection, source code transformation), OCL needs to be complemented/integrated with other tools/languages. The solution identified and implemented is the integration with the \texttt{JavaCC} environment, where the programmer retrieves nodes/properties executing
OCL queries with an approach similar to what done to executing SQL queries from any programming language.

### A.5 Tool Development Issues

![Example of OCL expression parse tree.](image)

The interpreter must integrate the knowledge of the OCL domain while being as independent as possible from the given target language domain, thus ensuring portability across different languages or programming paradigms. Furthermore, the mapping of structures or, more generally, the variable binding between the OCL domain space and the AST structures must be supported. As already mentioned, the Java programming language and Java tools (such as JavaCC) were identified as the environment for implementing the OCL interpreter. Java allows to write portable code, while providing powerful structures and algorithms via standard packages. Moreover, the reflection package allows to navigate a class structure known at runtime; this is particularly useful, as shown below, for implementing the OCL interpreter's navigation over the AST object model. Finally, efficiency concerns can be addressed, if needed, via Java virtual machines implementing an aggressive optimization, such as jrockit, or by compiling Java source into the target machine binary code (e.g., via Java compiler such as gcj).
There are several classes required to implement an interpreter, the most important class is the OclInterpreter class. The method OclInterpreter::execute takes as parameters:

- A node from the AST of the target language to be interpreted;
- A string, containing the query written in OCL language; and
- A symbol table.

The method returns an Object that may contain another AST node or a scalar value.

The target language domain is composed of a lexer and a parser. The latter produces, relying on the jjtree tool (part of JavaCC) the AST (consisting, as shown in Figure A.2, in a set of classes) of the source code analyzed.

The default jjtree AST structure has been properly modified, in order to provide each node with an attribute implementing the association with all its children.

The OCL domain is composed of:

- An OCL lexer and parser;
- A symbol table; and
- An attribute evaluator that works as an interpreter.

According to what described in [25] the symbol table binds variables to contained values. To ease the development task, the symbol table stores intermediate and final computation results corresponding to nodes of the OCL expression AST; it must also provide scope mechanism and binding. The binding functionality is also needed to map AST nodes into OCL symbols belonging to the expression to be evaluated. This, in turn, brings back to the need of defining an abstract representation for symbol table entries, corresponding to OCL expressions and sub-expressions. Working in Java this is readily available in that the Java Object class offers the method hashCode(). If two objects are equal according to the equals(Object) method, then calling the
hashCode method on each of the two objects must produce the same integer result, thus they are mapped, as required, into the same symbol table entry.

The interpreter semantics is implemented by an attribute evaluator: an eval method is provided for each node of the OCL grammar. Such function evaluates the node according to:

- The passed symbol table;
- A passed node (that, often, corresponds to the part of the OCL expression already evaluated), and
- The node local values.

Let us suppose we need to count the number of return statements inside a Java method m (see the function ReturnStatement in Figure A.4.2). A simplified version of the OCL expression parse tree is shown in Figure A.5. Terminal symbols are bold-faced, while some grammar nodes (in particular, those corresponding to the different precedence levels of the expressions), for sake of simplicity, are not reported.

From an high-level point of view, evaluating such expression means to sequentially evaluate all the subtrees of the node primaryExpression on the top of the tree, except the first (primaryExpression, that simply indicates the access to the node passed as parameter). In each case, an eval method is invoked.

The eval method of the first propertyCall takes as a parameter the Object returned by the previous invocation. When the terminal node matches to "methodBlock", the homonym attribute of the Object m is accessed and returned. This is implemented, as above mentioned, using the Java reflection package. It is worth noting that the terminal node is reached by recursive invocation, from each node, of the eval method for all its children.

The second eval method takes again an Object as a parameter (m.methodblock) and, when the terminal node matches to "descendants", a visitor is executed on the node in order to retrieve all its descendants.

The third eval method calls the eval of its child node (feature), which has a different behavior depending on the feature actually called. In this case (select),
it iterates on all nodes received as parameter, passing each one as parameter to its rightmost subtree, that evaluates a Boolean expression on the node. If the return value is true, the node is appended to the Sequence to be returned as a result to the parent node (primaryExpression).

Finally, the result of the select is passed to the eval method of the rightmost subtree. Similarly to the previous case, once the feature has been identified as a size, the eval returns the number of items contained in the Sequence received as parameter.

Other, more complex, OCL expression can be similarly evaluated. When an OCL feature defines an iterator variable or an accumulator, the scoped symbol table is used to keep track of the intermediate values.

![OCL interpreter package diagram](image)

Figure A.6: OCL interpreter package diagram.

The package diagram of the OCL interpreter is shown in Figure A.5. The **OCL Interpreter** package contains all public classes of the interpreter, i.e., all classes users can import in their own programs to use the interpreter itself. The **OCL Interpreter** package, on its own, relies on four other packages:

1. The **OCL Parser** package contains the JavaCC parser for the OCL languages and the classes implementing the AST of an OCL expression;

2. The **Semantics** package, implementing semantic actions of the interpreter;

3. The **OCL types** packages, implementing the OCL types (Collection, Sequence, Set, Bag, oclAny, Integer, String, Real, Boolean); and
4. The *OCL target language* package, that wraps the parser of any target language.

The object model of the target language AST is represented using a XML (Extensible Markup Language) file, therefore it is possible to change it/add new languages without affecting the rest of the interpreter.

```java
import java.util.Vector;
import java.io.*;
import oclinterpreter.*;
public class example {
    public static void main (String [] args){
        OclInterpreter oclint;
        Object result;
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        String expr;
        String filename = args[0];
        try {
            oclint = new OclInterpreter(OclInterpreter.JavaLang);
            while ((expr = input.readLine()) != null) {
                OclPreparedStatement oclprestmt = new OclPreparedStatement(expr);
                result = oclint.oclexecute(filename, oclprestmt);
                System.out.println("Result is:\n*result");
            }
        } catch (Exception e) {
            System.err.println("Error"+e.getMessage());
        }
    }
}
```

Figure A.7: Example of OCL interpreter use.

Finally, an example of how the interpreter can be used is shown in Figure A.5. Once an object of type `OclInterpreter` has been instanced, it is possible to ask the interpreter to execute an OCL statement. This can be done in four different ways, using different variants of the `oclexecute` method:

1. By preparing the statement (i.e., pre-parsing it) and then executing it on a target language source file (as shown in Figure A.5);

2. By preparing the statement and then executing it on the result set obtained from a previous query;

3. By directly executing an OCL expression contained in a string on a target language source file; or
4. By directly executing an OCL expression contained in a string on the result set obtained from a previous query.

As said, the above described mechanisms are very similar to the way an application interacts with a relational database executing SQL queries.

A.6 Conclusions and Work-in-Progress

It has been shown how OCL can be used for navigating and querying ASTs. This gives to the maintainer the following advantages:

1. The language used is not a proprietary language, but it is a de facto standard;

2. The grammar can be thought as an object model, thus supporting its visualization using common UML viewers/editors;

3. OCL revealed itself to be particularly effective for several tasks, such as browsing AST, computing metrics and clone detection; and

4. It is possible to decouple the query tool/language from the target language meta-model, therefore the tool can be easily extended to support new domains/languages.

Work-in-progress is devoted to implement a viewer allowing to browse an AST using an object diagram.
Bibliography


http://www.experimentalstuff.com/Technologies/JavaCC/.


[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.


ABSTRACT

Evolution Doctor: A Framework to Control the Evolution of Undocumented Software Systems

by
Massimiliano Di Penta

Advisor: Professor Giuliano Antoniol

Real world software systems undergo to repeated maintenance activities during their lifetime. Due to the market pressure and to the need to having back the system operational in the shortest time possible, these maintenance activities tend to introduce negative side effects. Some examples are the growth of the cloning percentage, the increase of library size, the presence of unused objects, the lost of source file organization and traceability links.

This thesis proposes a framework, named Evolution Doctor, to diagnose and cure such phenomena. First and foremost, the framework allows to analyze and predict several aspects of software system evolution (size, complexity, cloning). Secondly, the framework defines a set of methods and tools to diagnose and cure the problems; reorganize libraries, restructure the source file directory organizations, identify design patterns into object-oriented code, and recover traceability links.