A Syntactical Reverse Engineering Approach to Fourth-Generation Programming Languages Using Formal Methods

Majd Abdullatif Zohri Yafi

A thesis submitted for the degree of
Doctor of Philosophy

School of Computer Science and Electronic Engineering
University of Essex

May 2019

Abstract

Fourth-generation programming languages (4GLs) feature rapid development with minimum configuration required by developers. However, 4GLs can suffer from limitations such as high maintenance cost and legacy software practices.

Reverse engineering an existing large legacy 4GL system into a currently maintainable programming language can be a cheaper and more effective solution than rewriting from scratch. Tools do not exist so far, for reverse engineering proprietary XML-like and model-driven 4GLs where the full language specification is not in the public domain.

This research has developed a novel method of reverse engineering some of the syntax of such 4GLs (with Uniface as an exemplar) derived from a particular system, with a view to providing a reliable method to translate/transpile that system's code and data structures into a modern object-oriented language (such as C#).

The method was also applied, although only to a limited extent, to some other 4GLs, Informix and Apex, to show that it was in principle more broadly applicable. A novel testing method that the syntax had been successfully translated was provided using 'abstract syntax trees'.

The novel method took manually crafted grammar rules, together with Encapsulated Document Object Model based data from the source language and then used parsers to produce syntactically valid and equivalent code in the target/output language.

This proof of concept research has provided a methodology plus sample code to automate part of the process. The methodology comprised a set of manual or semi-automated steps. Further automation is left for future research.

In principle, the author's method could be extended to allow the reverse engineering recovery of the syntax of systems developed in other proprietary 4GLs. This would reduce time and cost for the ongoing maintenance of such systems by enabling their software engineers to work using modern object-oriented languages, methodologies, tools and techniques.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Why Reverse Engineering? . . . 1

1.1.2 Why Computer Languages? . . . 2

1.1.3 Research Focus and Method . . . 3

1.2 Key Concepts and Definitions . . . 4

1.2.1 Language Generations . . . 4

1.2.2 Reverse Engineering Concepts . . . 4

1.2.3 Grappling with Reverse Engineering Legacy Systems . . . 5

1.2.4 Domain Specific Languages . . . 6

1.3 Fourth-Generation Programming Languages . . . 6

1.3.1 Introduction to 4GLs . . . 7

1.3.2 Examples of 4GLs . . . 7

1.3.3 Uniface as a 4GL . . . 8

1.3.4 Uniface Compared to .NET . . . 8

1.4 Reverse Engineering and Research Approaches Used for 4GLs . . . 9

1.4.1 Design Recovery . . . 10

1.4.2 Re-engineering . . . 10

1.4.3 Code Translation . . . 11

1.4.4 Reverse Engineering Approaches for Correctness . . . 11

1.5 4GLs in Industry and the ESIS Case Study . . . 12

1.5.1 4GLs in Education and Industry . . . 12

1.5.2 Case study: University of Essex, ESIS and the Uniface 4GL . . . 13

1.5.3 Case Study: Essex University ESIS System . . . 14

1.6 Research Approach . . . 16

1.6.1 Research Direction for this Thesis . . . 16

1.6.2 Research Aims . . . 16

1.6.3 Research Objectives . . . 17

1.6.4 Contribution to Knowledge and Societal Benefits . . . 18

1.6.5 Technical Outcomes . . . 20
1.7 Research Questions . . . 20
1.8 Thesis Structure . . . 20

2 Literature Review **23**
2.1 4GL Literature . . . 23
2.2 Methods Used to Reverse Engineer 4GLs . . . 25
2.2.1 Knowledge-Based Software Engineering (KBSE) . . . 25
2.2.2 Design Recovery . . . 29
2.2.3 Purdue Compiler Construction Tool Set (PCCTS) . . . 31
2.2.4 Code Generation . . . 32
2.3 Supporting Techniques and Concepts . . . 32
2.3.1 Design Patterns . . . 33
2.3.2 Evolution of Abstract Syntax Trees . . . 34
2.4 Re-engineering/Reverse Engineering Legacy Systems Techniques . . . 36
2.4.1 Re-engineering - Surveys . . . 36
2.4.2 Re-engineering Existing Code to Enhance and Promote Code Reuse . . . 37
2.4.3 Re-engineering - Code Analysis and Transformation Tools . . . 38
2.4.4 Re-engineering - Cost and Complexity . . . 39
2.4.5 Re-engineering - Querying the Extracted Knowledge . . . 40
2.4.6 Re-engineering the Underlying Syntax . . . 41
2.4.7 Re-engineering Technology Dependent Tools . . . 42
2.4.8 Re-engineering for Reconstructing Design and Documentation . . . 43
2.4.9 Re-engineering for Recovering the Entity Relationship Modelling . . . 44
2.4.10 Re-engineering Using Parsers and Parser Generation Technologies . . . 45
2.5 Re-engineering/Reverse Engineering Legacy 4GLs - Research Gap . . . 46
2.6 Conclusion . . . 46

3 Methodology **48**
3.1 Deriving a Data Model from Extracted Uniface Code . . . 48
3.1.1 Uniface's Built-in Database . . . 48
3.1.2 Using Unified Modeling Language (UML) . . . 49
3.1.3 Modelling Approaches for this Research . . . 52
3.1.4 Using Relational Algebra . . . 52
3.2 Creating an Abstract View of the Uniface System's Behaviour . . . 53
3.3 Using Component-Based Methodologies Towards Reverse Engineering Uniface . . . 56
3.3.1 Introduction . . . 56
3.3.2 Component-Based Software Engineering (CBSE) . . . 56
3.3.3 Advantages of Component-Based Design (CBD) . . . 58
3.3.4 Incremental Development - Advantages, Principles and Techniques . . . 60

3.3.5 Adapting Existing Tools or Building Bespoke Tools? . . . 63
3.4 Compiler Framework Design . . . 64
3.4.1 Introduction to Compilers, Parsers and Lexers . . . 64
3.4.2 Parsers . . . 65
3.4.3 Abstract Syntax Trees (AST) . . . 65
3.4.4 Parser Techniques . . . 66
3.4.5 Grammars . . . 67
3.4.6 Parser Generator Tools . . . 68
3.5 Determining the Syntactical Correctness of Translated 4GLs . . . 70
3.6 Conclusion . . . 72

4 Implementation I - Extraction, ER and Relational Algebra Modelling 73
4.1 System Architecture . . . 74
4.2 Extracting Uniface XML-like Code and Static Model Extraction . . . 75
4.2.1 Extracting Clean XML Data from Uniface . . . 75
4.2.2 Static Model Extraction . . . 75
4.3 Using Relational Algebra to Represent the ER Model . . . 78
4.4 Parsing Uniface Data and Meta-Data . . . 80
4.4.1 Can HTML Tools parse Uniface XML-like Code? . . . 80
4.4.2 Can XML Tools Parse Uniface XML-like Code? . . . 80
4.4.3 Can EDOM Tools Parse Uniface XML-like Code? . . . 81
4.5 Encapsulated Document Object Model (EDOM) . . . 81
4.5.1 Introduction to EDOM . . . 81
4.5.2 Using EDOM for Large Files . . . 82
4.5.3 Parser: Tags and Nodes . . . 82
4.5.4 Parser Processing Logic . . . 84
4.5.5 Example of Parser Processing XML . . . 84
4.6 Why Unicode is Needed for the Uniface Character Set . . . 85
4.7 Conclusion . . . 87

5 Implementation II - Reverse Engineering Uniface Schema 88
5.1 Reverse Engineering Uniface Schema Using Document Type Definition (DTD) Form 88
5.1.1 Introduction to DTD Schema . . . 88
5.1.2 Uniface Document Type Definition Syntax . . . 90
5.2 Reverse Engineering Uniface Schema Using XML Schema Definition (XSD) Form 93
5.2.1 Introduction to XSD Schema . . . 93
5.2.2 XML Schema Definition Syntax . . . 94
5.3 Conclusion . . . 98

6 Implementation III - Extracting Uniface Elements 99

6.1 Splitting up Large XML Files - Extracting TABLE Elements 99
6.1.1 Need for Dividing a Large XML File 99
6.1.2 Splitting the XML File 100
6.1.3 Extracting 'OCC' Elements and Finding TABLE Elements 102
6.2 Exporting XML to SQL 105
6.3 Parsing Techniques for Programming Languages 105
6.3.1 Parsing Using Regular Tree Grammar (RTG) 106
6.3.2 Parser Algorithms 106
6.3.3 Using LL(k) Parsers 107
6.4 Meta-syntax Notation 107
6.4.1 Meta-syntax Notations as Formal Input 107
6.4.2 Using SQL 'CREATE TABLE' as an Example 107
6.4.3 Example: Parsing SQL 'CREATE TABLE' 110
6.4.4 Parsing Uniface Grammar 112
6.5 Conclusion 112

7 Implementation IV - Grammar Formalism 114
7.1 Principles of Grammar Formalism 114
7.1.1 Domain Specific Languages (DSL) Need Regular Expressions + Extensibility 114
7.1.2 Modular Grammar 115
7.1.3 Grammar Engineering for Uniface 118
7.1.4 Grammar Correctness 119
7.1.5 Algebraic Specification Formalism (ASF) 121
7.1.6 Syntax Definition Formalism 122
7.1.7 Parsing Ambiguous Grammar - An Example 122
7.1.8 Template Meta-Programming and Templates 125
7.1.9 Using BNF/EBNF for Syntax Notation 126
7.1.10 Syntax Diagrams 127
7.1.11 Overview of System as Implemented 127
7.2 Data Flow from Front-end to Back-end Layer 132
7.3 Conclusion 133

8 Using Parser Generators to Implement Uniface Grammar 134
8.1 S.O.L.I.D. Principles 134
8.2 .NET Framework and C# Programming Language 135
8.3 Implementation in Gold Parser 136
8.3.1 Parser Operation 136
8.3.2 Gold Parser 137
8.3.3 Functions and Parameters 139
8.3.4 Tokeniser 139

8.3.5 Multi-language Support . . . 139
8.3.6 Table Output as .egt File . . . 139
8.3.7 Main Screen - User Interface . . . 141
8.3.8 Analysis of Input . . . 142
8.3.9 Implementing Grammar Rules in Gold Parser . . . 143
8.3.10 Producing a Derivative Tree . . . 145
8.3.11 Testing Grammar . . . 147
8.3.12 Gold Parser Usage in this Research . . . 148
8.4 Implementation in ANTLR . . . 149
8.4.1 ANTLR Parser . . . 149
8.4.2 ANTLR Parser Basic Example . . . 149
8.4.3 Parser Backtracking . . . 150
8.4.4 Implementing Grammar Rules in ANTLR . . . 150
8.4.5 Debugging ANTLR Output . . . 156
8.4.6 ANTLR 'Listener' Output File . . . 156
8.4.7 Augmented Transition Network (ATN) Visualisation . . . 157
8.4.8 Call Graph Visualisation . . . 157
8.4.9 ANTLR 'Base Visitor' Partial Class . . . 157
8.4.10 Using ANTLR Output . . . 158
8.4.11 Traversing Grammar Rules . . . 161
8.5 Comparison Between Gold Parser and ANTLR . . . 162
8.6 Evaluation of Gold Parser and ANTLR Algorithm Strategy . . . 162
8.7 Conclusion . . . 163

9 Extending from Uniface to Other 4GLs 165
9.1 Reverse Engineering Other 4GLs . . . 166
9.1.1 Informix . . . 167
9.1.2 Apex . . . 167
9.2 Building a New Mini-4GL Language . . . 167
9.3 Reverse Engineering the New Mini-4GL Language . . . 173
9.4 Conclusion . . . 176

10 Results 178
10.1 Language Syntax and Semantics . . . 179
10.2 Comparing Abstract Syntax Trees for Input and Target Languages . . . 181
10.2.1 Tree Search Comparison Methods . . . 182
10.2.2 Abstract Syntax Trees and their Derivations . . . 182
10.3 Using Chunking Trees to Compare ASTs for Source Language and Target Language Syntax . . . 185
10.4 Summary of Results: Comparison Runs . . . 190

10.4.1 Summary of Results - Uniface Comparison Runs 190
10.4.2 Summary of Results - Informix 4GL Comparison Runs 192
10.4.3 Summary of Results - Apex Comparison Runs 196
10.4.4 Summary of Results - Mini-4GL Comparison Runs 202
10.5 Evaluation of Results 203
10.6 Measuring Transformation Scope 204
10.7 Conclusion 205

11 Conclusions 206
11.1 Recapitulation of Reverse Engineering, ESIS Case Study and Research 206
11.2 Answering the Research Questions 209
11.2.1 Main Research Question 209
11.2.2 Sub-Research Question 1 210
11.2.3 Sub-Research Question 2 210
11.3 Summary of Contributions to Knowledge 211
11.3.1 Applying CBSE Methodology to a Reverse Engineering Model-Driven Proprietary 4GL 211
11.3.2 Utilising EDOM for Processing XML-like Code into a List of Valid XML Files 211
11.3.3 Novel Grammar Rules for Uniface 211
11.3.4 Use of Parser-Parser and Parser Generators to Translate a XML-like Proprietary 4GL into a 3GL 212
11.3.5 Using Chunking Trees to Syntactically Compare Programming Languages 212
11.4 How the System Could be Improved/Completed 213
11.4.1 More Complete Set of Grammar Rules for Uniface 213
11.4.2 Automation of the System 213
11.5 Lessons Learned 214
11.6 Future Work 214
11.7 Concluding Remarks 215

A Output Code in C# from Input Code in Uniface 216

B Input - Uniface Code from ESIS System (Extract) 226

C C# Code to Transpile Uniface into C# 239

D Generated Parsing Table in Gold Parser 242

E Generating Abstract Syntax Tree Using ANTLR for Other 4GLs 273
E.1 Informix 273
E.1.1 Informix Sample Code Snippet 273
E.1.2 Informix Abstract Syntax Tree (AST) 274

E.1.3 Informix Snippet into C# Output 274
E.2 Apex 275
E.2.1 APEX Sample Code Snippet 275
E.2.2 APEX Abstract Syntax Tree (AST) 275
E.2.3 APEX Snippet into C# Output 276

F Programming Languages Timeline Between 1950 and 2003 277

G Uniface TABLE Element 285

H C# Code to Transpile Mini-4GL into C# 287

I Using String Template Methods to Implement Template Meta-Programming 296

Bibliography 309

Associated Publications

Portions of the work detailed in this thesis have been presented in national and international peer-reviewed publications, as follows:

PRINCIPAL AUTHOR

- Yafi, M. Z., and Fatima, A., "Syntax Recovery for Uniface as a Domain Specific Language." 2018 UKSim-AMSS 20th International Conference on Computer Modelling and Simulation (UKSim), Cambridge, United Kingdom, pp. 61–66, IEEE, 2018.

CO-AUTHOR

- Alzahrani, A. A. H., Eden, A. H. and Yafi, M. Z., 'Conformance Checking of Single Access Point Pattern in JAAS using Codecharts', World Congress on Information Technology and Computer Applications 2015 (WCITCA), Hammamet, Tunisia, pp. 1-6, IEEE, 2015.

- Alzahrani, A. A. H., Yafi, M.Z., and Alarfaj, F., "Some Considerations on UML Class Diagram Formalisation Approaches." World Academy of Science, Engineering and Technology, International Science Index 89, International Journal of Computer and Information Engineering 8.5 (2014), pp. 741-744

- Alzahrani, A. A. H., Eden, A. H., and Yafi, M. Z., "Structural Analysis of the Check Point Pattern." in 2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE), Oxford, United Kingdom, pp. 404-408, IEEE, 2014.

List of Figures

1.1 Number of Public Programming Languages created between 1956 and 2003. Author's graph based on data from Lévénez [1] . . . 2
1.2 Model-Driven Development versus .NET and Java (Source: Uniface White Paper [2]) 9
1.3 Languages Interoperability. (Bissyandé et al. [3]) . . . 15

2.1 Reverse Engineering Using KBSE Concepts - according to Canfora et al. [4] . . . 27
2.2 Phrase Structure Grammar Tree - Floyd [5] . . . 35
2.3 Using Querying as the Gate Layer to Perform Various Re-engineering Activities - Sartipi et al. [6] . . . 41

3.1 UML History 1990-2008 (Zockoll et al. [7]) . . . 50
3.2 Uniface Activities (Uniface, 2000) [8]. . . . 57
3.3 Component-Based Development Process (CBDP). (Uniface) [8] . . . 59
3.4 Parser Structure: Lexer, Tokeniser, and Parse Tree (Author's Diagram) . . . 65
3.5 Parse Tree, Abstract Syntax Tree . . . 66

4.1 System Architecture - High Level Generic . . . 74
4.2 Uniface Relationship Builder Tool . . . 77
4.3 ER model for Student Data Entities ASR_DET and PR_MM . . . 78
4.4 Student Data Entities ASR_DET and PR_MM - Primary Key SQL Query . . . 78
4.5 Student Data Entities ASR_DET and PR_MM - Primary Key Result-set . . . 78
4.6 JSON - Transformed Query . . . 79
4.7 JSON Syntax . . . 80
4.8 XML Characters Numbered by Column Position . . . 85

5.1 High Level Schematic of XML Interpreters. Taken from Yafi and Fatima [9] . . . 90
5.2 Uniface Schema in Document Type Definition (DTD) Format. Taken from Yafi and Fatima [9] . . . 91
5.3 Uniface Schema in XML Schema Definition (XSD) Format. Taken from Yafi and Fatima [9] . . . 94
5.4 XSD Simple Data-Types Source: XML Schema Types, w3.org [10] . . . 97

6.1 XML File Splitter: Code and Output 101
6.2 Uniface TABLE Element Structure 102
6.3 The Process of Extracting TABLE and 'OCC' Elements Taken from Yafi and Fatima [9] 104
6.4 SQL Grammar - Create Table 109
6.5 SQL Tree - Create Table 111
6.6 Uniface If-Block Parse Tree 112

7.1 Ambiguous Grammar: Expression Paths 123
7.2 Steps for Generating Scannerless Generalised LR Parser (SGLR) and Parse Tree 124
7.3 Uniface If-Block 126
7.4 General Overview of the System's Components Grouped by the Relevant Layer . 128
7.5 Topological Architecture of the System 129
7.6 Object Relational Mapping for the Middleware Layer 131

8.1 Gold Parser - Deterministic Finite Automaton for a Sample Input 138
8.2 Gold Parser - Deterministic Finite Automaton Screen 138
8.3 Gold Parser - Tool Main Screen with Example Code for Parsing Uniface 140
8.4 Gold Parser - Analyser Screen 142
8.5 Gold Parser - LALR Table Generation Screen 146
8.6 Gold Parser - Test Window 147
8.7 Gold Parser - Test Window - Input Sample Parsed Successfully 148
8.8 ANTLR: String Rule Railroad Representation 154
8.9 ANTLR: If-Rule Railroad Representation 155
8.10 String-Rule ATN Visualisation 158
8.11 If-Rule ATN Visualisation 159
8.12 String-Rule Call Graph Visualisation 160
8.13 UML Representation of the Visitor Design Pattern. Taken from W3s Design [11] 161
8.14 LePUS3 Representation of Visitor Design Pattern From Alzahrani, M. Yafi et al. [12]161

9.1 MPS Mini-4GL Language Editor 168
9.2 MPS Grammar Rules Editor - The Grammar for the Main Screen of Mini-4GL . 169
9.3 MPS Grammar Rules Editor Project_Component 169
9.4 MPS Behavioural Editor for Mini-4GL 170
9.5 TextGen for the HTML Document 171
9.6 The 'Concept' that Represents the Standard HTML Structure 171
9.7 Transformation Menu for Project_Component 172
9.8 HTML Output for the New Mini-4GL Language 173
9.9 Source Editor Code Generation Process 174
9.10 Projectional Editor Code Generation Process 174

9.11 Template Programming to Generate PharmacyX Output 176

10.1 Abstract Syntax Tree for a Sample C# Application 180
10.2 Minimum Complete Tree (MCT) 184
10.3 Path-enclosed Tree (PT) 184
10.4 Chunking Tree (CT) 185
10.5 Run U1: Chunking Trees to Compare Uniface (Tree A - top) to C# (Tree B - bottom) Chunking Trees 186
10.6 Informix Function Definition (top) Chunking Tree - Compared to C# (bottom) . 193
10.7 Informix For-Loop Railroad [13] 194
10.8 Informix Inner For-Loop (top) Chunking Tree - Compared to C# (bottom) . . . 195
10.9 Apex Attribute (top) Chunking Tree - Compared to C# (bottom) 197
10.10 Apex Class and Function Definition (top) Chunking Tree - Compared to C# (bottom) 198
10.11 Apex Attribute (top) Chunking Tree - Compared to C# (bottom) 200
10.12 Apex Attribute (top) Chunking Tree - Compared to C# (bottom) 201
10.13 Run M4 ‘th’ element: Chunking Trees to Compare Mini-4GL (Tree A - top) to HTML (Tree B - bottom) Chunking Trees 203

11.1 Processes Within System of Transforming Uniface by Level of Automation 214

E.1 Informix AST for Sample Code 274
E.2 AST for Apex Sample Code 275
E.3 AST for Apex Sample Code Into C# 276

List of Tables

1.1 List of COBOL Dialects (Van Den Brand et al. [14]) 3

2.1 Comparison of 3GL and 4GL Traits according to Ketler and Smith [15] 24

4.1 Entities ASR_DET and PR_MM, their Keys and Attributes Types 77
4.2 Comparison between Unicode and non-Unicode Characters 86

7.1 OData Standards 133

10.1 Run U1: Chunking Tree Comparison Uniface to C# - Explanation of Labels . . . 188
10.2 Summary Results Runs U1 - U10: Chunking Tree Comparisons Uniface to C# . 191
10.3 Summary Results Runs I1 - I2: Chunking Tree Comparisons Informix to C# . . 193
10.4 Summary Results Runs I1 - I7: Chunking Tree Comparisons Informix to C# . . 196
10.5 Summary Results Runs A1 - A10: Chunking Tree Comparisons - Classes and Functions - Apex to HTML 199
10.6 Summary Results Runs A1 - A4: Chunking Tree Comparisons - Method Definition Part 1 - Apex to C# 200
10.7 Summary Results Runs A1 - A8: Chunking Tree Comparisons - Method Definition Part 2 - Apex to C# 202
10.8 Summary Results Runs M1 - M5: Chunking Tree Comparisons Mini-4GL to HTML 203

List of Abbreviations

Abbreviation	Full Form
ABAP	Advanced Business Application Programming
AI	Artificial Intelligence
AJAX	Asynchronous JavaScript and XML
ANTLR	ANother Tool for Language Recognition
APG	ABNF Parser Generator
AQL	Architectural Query Language
ARIS	Architecture of Integrated Information Systems
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
ATN	Augmented Transition Network
B&B	Branch-and-Bound
BNF	Backus-Naur Form
BSON	Binary JSON = Binary JavaScript Object Notation
CASE	Computer-Aided Software Engineering
CAISE	Computer Aided Information Systems Engineering
CBD	Component-Based Design
CBDP	Component-Based Development Process
CBSE	Component-Based Software Engineering
CCT	Context-sensitive Chunking Tree
CGT	Compiled Grammar Table
COM	Component Object Model
CPT	Context-sensitive Path Tree
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
CRUDQ	Create, Read, Update, Delete, Query
CSS	Cascading Style Sheet(s)
CT	Chunking Tree

Abbreviation	Full Form
DBCS	Double-byte Character Sets
DFA	Deterministic Finite Automaton
DLG	DFA-based Lexical analyser Generator
DML	Data Manipulation Language
DOM	Document Object Model
DPD	Design Pattern Detection
DSL	Domain Specific Language
DSP	Dynamic Server Pages
DTD	Document Type Definition
EBNF	Extended Backus-Naur Form
EDA	Event-Driven Architecture
EDM	Entity Data Model/Modelling
EDOM	Encapsulated Document Object Model
EDSL	Embedded Domain Specific Language
EF	Entity Framework
ERM	Entity–Relationship Model
ERP	Enterprise Resource, Planning
ESIS	Electronic Student Information System
FSM	Finite State Machine
GLL	Generalised Left Left
GLR	Generalised Left Right
GML	Graph Modelling Language
GPLEX	Gardens Point LEX
GReQL	Graph Repository Query Language
GSS	Graph-Structured Stack
IDEs	Integrated Development Environments
IID	Iterative and Incremental Development
IL	Intermediate Language
IoT	Internet of Things
ISO	International Organisation for Standardisation
JAD	Joint Application Development
JIT	Just-In-Time
JSON	JavaScript Object Notation
KBSE	Knowledge-Based Software Engineering
KPI	Key Performance Indicators
LALR	Look-Ahead LR parser
LDAP	Lightweight Directory Access Protocol

Abbreviation	Full Form
LL	Left-to-right, Leftmost derivation
LLLPG	Loyc LL(k) Parser Generator
LP	Logic Programming
MCT	Minimum Complete Tree
MDD	Model-Driven Development
MOF	Meta-Object Facility
MOOSE	Multiphysics Object-Oriented Simulation Environment
MPS	Meta-Programming System
MVP	Minimum Viable Product
OLTP	Online Transaction Processing
OMG	Object Management Group
OMT	Object-Modelling Techniques
OOAD	Object-Oriented Analysis and Design
OOSD	Object-Oriented Software Design
OOSE	Object-Oriented Software Engineering
ORM	Object Relational Mapping
PCCTS	Purdue Compiler Construction Tool Set
PEG	Parsing Expression Grammar(s)
POC	Proof of Concept
POP	Post Office Protocol
PSG	Phase Structure Grammar
PT	Path-enclosed Tree
RACI/RACI Charts	Responsible, Accountable, Consulted and Informed
RE	Regular Expressions
RTE	Round-trip Engineering
RTG	Regular Tree Grammar
SAX	Simple API for XML
SCM	Software Configuration Management
SDF	Syntax Definition Formalism
SDLC	Software Development Life Cycle
SGLR	Scannerless Generalised LR parser
SOA	Service-Oriented Architecture
SoSR	Service-Oriented Software Re-engineering
SQL	Structured Query Language
SRE	Software Reverse Engineering
SSH	Secure Shell Protocol
SWEBOK	Software Engineering Body of Knowledge

Abbreviation	Full Form
TMP	Template Meta-Programming
UI	User Interface
UML	Unified Modeling Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSI	XML Schema Instance

Chapter 1

Introduction

In this document, abbreviations have been spelt out on first usage, and a complete list may be found immediately preceding this section.

This thesis demonstrates that the syntax of legacy software code in a proprietary 4th generation programming language (4GL) can be reverse engineered into an object-oriented model for ease of comprehension and maintainability.

1.1 Motivation

1.1.1 Why Reverse Engineering?

Programming languages vary in their features and abilities. The discipline of Software Engineering includes a sub-discipline which studies the traits of these languages and the methods used to craft a new language. This process is called forward engineering whereas the sub-discipline that studies disassembling programs written in a target language and dissecting them, is known as reverse engineering.

"Software Reverse Engineering (SRE) is the practice of analysing a software system, either in whole or in part, to extract design and implementation information" according to Cipresso and Stamp [16]. This process is beset with challenges due to the size of programming code and data that the legacy system might have contained at the time it was considered for re-engineering. Legacy systems are defined as "the large software systems that we don't know how to cope with but that are vital to our organization", according to Bennett [17]. In addition, legacy systems do not maintain adequate, accurate or up to date documentation in most cases. Reverse engineering exploits these challenges and presents automated or semi-automated solutions to recover the system design (Chikofsky et al. [18], Harrison and Berglas [19], Sadiq and Waheed [20]) and

visualise the system's meta-data and database's schema by employing software visualisation theory (Diehl [21], Ball and Eick [22], Price et al. [23]).

1.1.2 Why Computer Languages?

Computer programming languages are a crucial subject in software engineering. According to Lämmel and Verhoef [24] there are more than 500 languages available commercially or publicly and about 200 proprietary languages that are found by companies to be used internally Roetzheim [25]. Figure 1.1 plots the data listed in appendix F taken from Lévénez [1], and shows the the number of publicly available programming languages that were created between years 1950 and 2003 Lévénez [1] not including the number of dialects each language may have (van den Brand et al. [14]) and (Weinberg [26]).

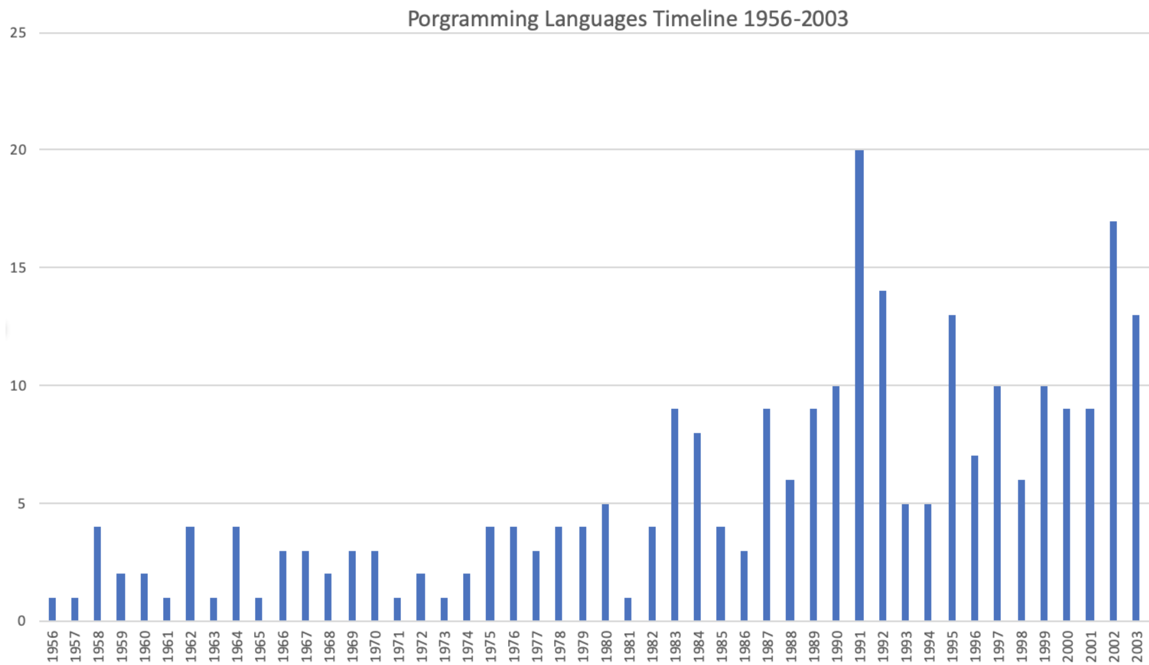

Figure 1.1: Number of Public Programming Languages created between 1956 and 2003. Author's graph based on data from Lévénez [1]

As an example of dialects, table 1.1 lists COBOL language dialects which extend the original language compilers' features and inherit the original COBOL syntax.

Jones has supported the idea, that investing in programming languages, tools and grammar editors may support and improve the productivity of software maintenance teams in the long run, as it has been observed that companies spend the maintenance budget on enhancing the assets that rely directly on the language grammar [27].

COBOL Dialects		
ACUCOBOL-GT	COBOL-IT	COBOL/2
DEC COBOL-10	DEC VAX COBOL	DOSVS COBOL
Fujitsu COBOL	Hitachi COBOL2002	HP3000 COBOL/II
IBM COBOL SAA	IBM COBOL/400	IBM COBOL/II
IBM Enterprise COBOL	IBM ILE COBOL	IBM OS/VS COBOL
ICL COBOL (VME)	isCOBOL	Micro Focus COBOL
Microsoft COBOL	Realia COBOL	Wang VS COBOL
UNIVAC COBOL	Unisys MCP COBOL74	Unisys MCP COBOL85
Unix COBOL X/Open	Veryant isCOBOL	Visual COBOL
Tandem SCOBOL	Ryan McFarland COBOL	Tandem COBOL85
Ryan McFarland COBOL-85		

Table 1.1: List of COBOL Dialects (Van Den Brand et al. [14])

1.1.3 Research Focus and Method

This research has focused on the Uniface 4GL syntax as implemented using the University of Essex's Essex Student Information System (ESIS). This was used to provide an extract of Uniface syntax as shown in appendix B. Any confidential information held in the databases has been omitted or redacted from this research.

Parts of the method, such as generating Abstract Syntax Trees (AST) as used in this research to translate the ESIS Uniface syntax, were also unsuccessfully attempted using SQL to retrieve the design from the Uniface meta-database.

The author's full method, was also applied to other 4GLs to a limited extent, the other 4GLs covered being Informix and Apex. Grammar rules were generated for each language as well as an abstract view of the syntactical structure for each.

This research has chosen to invent a new 4GL using forward engineering methods. The research then used the same techniques, as more fully discussed and applied on Uniface, to successfully reverse engineer on a very limited basis, some other existing 4GLs (Informix and Apex) to create C# and a new Mini-4GL to create code in HTML.

Languages such as Informix and Apex have been widely-used, well documented and standardised whereas Uniface is an exception in terms of lacking in areas such as documentation, syntactical grammar rules, and semantic aspects.

1.2 Key Concepts and Definitions

1.2.1 Language Generations

Although this thesis is dealing with the issues of reverse engineering 4th generation programming languages (4GLs), it is important to be aware what the different language 'generations' from 1GL to 6GL refer to.

Programming languages have evolved from first-generation programming languages to the sixth-generation.

First-generation programming languages were created in binary to control the hardware directly.

Second-generation language were also low-level languages, that present mnemonics to extricate developers from writing code in zeros and ones. In addition, they offer a syntactic structure and dynamic memory space allocation, Abel [28].

Third-generation programming languages were mostly procedural, but the concept of object-oriented programming had been created in some of these languages. For example, Pascal implemented the object-oriented model later in its life.

Fourth-generation programming languages are domain specific languages with higher abstraction in comparison to the previous generations. These languages are both functional and also focused around database rich domains. To the best of the author's knowledge there has not been any modelling technique to visualise them. This underlies the main motivation for this research.

Fifth-generation programming languages are more focused on solving a specific problem rather than writing an application that runs an algorithm. This technology is commonly used with artificial intelligence applications.

Sixth-generation programming languages mimic spoken languages to instruct the computer to perform a specific action.

1.2.2 Reverse Engineering Concepts

Software reverse engineering has been defined as

> "The process of extracting software system information (including documentation) from source code" (IEEE Std 1219-1993 [29]).

The methodology set out in this thesis aligns with that definition in that it

- extracts valid source code from the system being reverse engineered (a 4GL),
- derives design data from it, as far as possible,
- visualises the extracted design information using appropriate methods,

- transforms the syntax of the original system's code into valid code in another language (a 3GL).

1.2.3 Grappling with Reverse Engineering Legacy Systems

Understanding legacy systems can be hard. This could be due to the ripple effect resulting from changing just one part of the system, or any of the following factors:

- a lack of understanding of the legacy architecture,
- the poor naming of legacy components and variables,
- an absence of testing or clear set of tests to run against a static set of test data (earlier or at point of making changes) and
- undocumented business or technical decisions taken during at inception and over the years since then.

Mohagheghi et al. found that any or all of these could hinder developers from attempting to make significant changes. Therefore, it has been best practice to carry out only small changes to them especially on systems with any safety critical software elements [30]. Furthermore, due to the complexity that is involved, in ensuring that any change or ‘fix’ does not uncover or create new problems or instabilities, companies have been deterred from exploring stable systems that work in production.

During the 1990's reverse engineering became more popular along with advancements in software technology Davis and Alken [31], Müller et al. [32]. Kadam et al. [33] surveyed the available reverse engineering tools and their continuing advances and compiled a list of existing reverse engineering applications. They grouped these applications into three main categories:

- Knowledge extraction tools,
- Cyber security domain tools for reversing malware, viruses and other detrimental and malicious scripts,
- Recovery tools that recover erased data from storage or recover information which went missing after the system had been built and delivered.

Reverse Engineering tools are also used by SRE companies themselves on their competitor's tools, investigating similar software features in the market.

The significance of the research in this thesis has been to provide a novel method for syntactical reverse engineering of software to which one or more of the following apply:

- Closed-source systems;
- Domain specific languages;

- Non-documented systems;
- Difficult to maintain systems;
- Non-documented language semantics;
- Scattered source code;
- Absence of any/all of disassemblers, debuggers, system monitoring tools, decompilers.

This thesis's research has used novel LL(k) parsing techniques (Sippu and Soisalon-Soininenet [34]) based on a formal grammar notation written in Extended Backus-Naur Form (EBNF). They were represented using syntax diagrams (railroad diagrams), Wirth [35], for translating the extracted source code into interfaces and methods in the targeted languages.

subsectionDomain Specific Languages - A Brief Introduction

According to Fowler [36], domain specific languages (DSL) are defined as "a computer programming language of limited expressiveness focused on a particular domain". A DSL aims to find a solution for existing problems in a specific domain such as: education, finance, or computer tools and deliver the needed tools to address as many requirements as possible in this particular domain. However, a DSL is limited to its domain and may not be used for other purposes.

1.2.4 Domain Specific Languages

Domain Specific Languages were introduced and defined in section 1.2.3 (Grappling with Reverse Engineering Legacy Systems).

Domain Specific Languages (DSL) are programming languages designed to solve a set of problems in a particular domain. For instance, in the domain of software testing, the language 'Gherkin', dos Santos and Vilain [37], was created to test the behaviour of a target software system. Gherkin is still used by automation test developers to write commands that simulates the user behaviour. The steps in Gherkin are then syntactically accessible to business analysts who are not qualified as coders to write hardcore applications.

'Gradle' Muschko [38] is another example of a DSL. Gradle is a build automation system language that supports building multiple large projects simultaneously. Gradle optimises the build process by identifying which artefacts are up to date and which ones need to be be rebuilt.

DSLs are typically rich in features to accommodate a specific domain requirements. However, DSLs are not designed to assist developers performing any operation beyond its particular domain.

1.3 Fourth-Generation Programming Languages

Fourth-generation languages have been the focus for this thesis's research so are discussed in more detail, with examples, below.

1.3.1 Introduction to 4GLs

Fourth-generation programming languages (4GLs) are DSLs that provide a higher level of abstraction of the computer hardware than the the third-generation programming languages they replaced. C, C++, C#, Java are examples of 3GLs. Some languages like Python are considered to be advanced third-generation languages because they can be extended with 4GL libraries to perform additional tasks. 4GLs offer more semantic properties and greater implementation power.

1.3.2 Examples of 4GLs

'Nomad' Rawlings [39]: Is a relational database managed by a 4GL language for data management, application development, data manipulation and reporting. The following example shows a basic command written in Nomad to increase the salary field by 6% when the average value of the 'RATING' field is equal or greater than 7

$$CHANGE\,ALL\,SALARY = SALARY * 1.06\,WHERE\,AVG(INSTANCE(RATING))\,GE\,7$$

'MAPPER' [40]: Is a cross-platform 4GL designed for data processing. When MAPPER was introduced it eliminated the intricacies of the methods used for processing data. MAPPER organises data into series of 'cabinet' and 'drawer' where developers can keep the reports. Each 'drawer' can keep up to 5000 records and each cabinet can store 8 'drawer's. MAPPER features are built-in and can be invoked with keywords. For instance, $@ADR$ command stores a report permanently. The 3GL equivalent implementation of this command can be written but involves writing a complicated program.

'Uniface' [41]: This is a 4GL designed to support a range of run-time environments such as .NET, JAVA-EE, and SOA. Uniface is database independent. It is capable of sourcing data from a variety of database engines such as ORACLE, MS SQL, and IBM DB2. In addition, it supports different file types such as text files, excel files and comma delimited files. Moreover, it supports file systems such as POP, LDAP, ActiveX and Component Object Model (COM).

Uniface is used to build enterprise applications. Uniface simplifies building forms, server pages, and reports. Forms are screens that support data interoperability between the server and client in a network environment. The server pages are web pages which are served from the cluster where Uniface is hosted. Reports are data presentation views that can be customised and presented in adherence to and compliance with the business requirements.

JHMore examples are: Cerner CCL (Sandgather [42]), FOCUS, RAMIS, and SQL, (Miller [43]), ADS/ONLINE, APPLICATION FACTORY, DATATRIEVE, IDEAL, INTELLECT, MANTIS, MIMER, NATURAL, NOMAD2, RAMIS II, SYSTEM W, USE-IT (Martin [44]), Informix-4GL (Kipp [45]), and the xBase group of languages (Efftinge et al. [46]).

Fourth-generation programming languages (4GLs) renovate third-generation programming lan-

guage functions into functioning keywords. There are common traits among those keywords. Firstly, they cover applications whereby each of those keywords encapsulates a set of atomic instructions which could be written in lower level language as lengthy procedures. Secondly, those keywords are as close to natural language words as they can be. Thirdly, 4GLs were designed as procedural languages so modelling them using class diagrams or sequence diagrams is not feasible.

1.3.3 Uniface as a 4GL

According to a Uniface White Paper [2], Uniface is a model-driven environment and Integrated Development Environment (IDE) that claims to expedite the build process for mission-critical enterprise applications. The Uniface IDE at the University of Essex runs under Microsoft's Windows operating system. Uniface uses a run-time engine both to deploy and interpret the code and also to manage the connection and access to its database. The deployment is operating system agnostic. Therefore developers reap the benefits offered by Uniface system to deploy the final product on Windows, Linux, Unix, IBM, or HP platforms. In addition, database admins are offered a choice from a variety of built-in database drivers to retrieve data from a range of database engines.

Developers use the IDE to design the user interface, select data sources, and compile applications with Uniface. The Uniface programming language is used to write the application code. Uniface language is a high level language that is syntactically categorised as a fourth-generation programming language.

Uniface claims that it is designed to solve the experience gap between popular languages such as .NET and Uniface (Uniface white paper [2]). Cross-training may bridge the gap but additional costs may be incurred.

1.3.4 Uniface Compared to .NET

The main difference between .NET and Uniface is that the latter adopts Model-Driven Development (MDD) which is promoted by Uniface in its white paper [2] as a productivity advantage. Figure 1.2 illustrates the concept.

According to Sriparasa, in addition to the support for Dynamic Server Pages (DSPs), Uniface manages Asynchronous JavaScript and XML (AJAX) calls automatically [47]. Working with DSPs and AJAX reduces overhead costs by reducing the time for developers to learn alternatives such as SQL adaptors and amalgamating scripts to establish the connection with the database provider and exchange data between the client and the server in the background without blocking the user interface (UI).

The language syntax of Uniface is another dissimilarity between the two languages. Unlike .NET, which is a strongly typed language, Uniface variables are loosely typed. Variables do not strictly

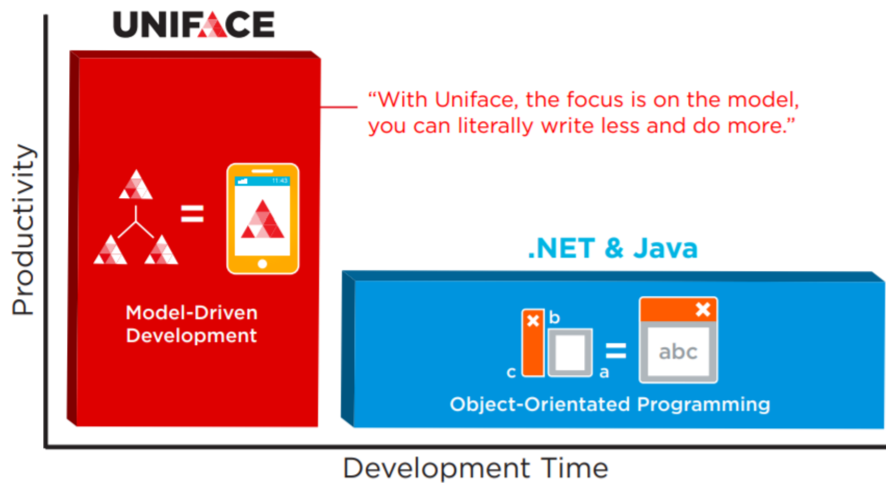

Figure 1.2: Model-Driven Development versus .NET and Java (Source: Uniface White Paper [2])

have to be declared and variable types are cast implicitly (i.e. Uniface changes variables types contextually).

In 4GLs such as the language Uniface studied for this author's research, the code generation process covers the implementation for the language's keywords and built-in functions.

The code generation process for this research, has been managed by a novel set of processes that takes Uniface 4GL code syntax as an input and generates the equivalent C# code syntax as an output.

For example, Uniface uses the keyword 'RETRIEVE' to retrieve data from the database. The 'RETRIEVE' function acts on the user model. The user model defines the database name, table name, and fields in the table where the data is stored. Internally Uniface generates the code that determines the appropriate connection for the database type and sends a query to read the data, and finally puts the data on the screen in the relevant user interface fields.

1.4 Reverse Engineering and Research Approaches Used for 4GLs

Researchers have approached 4GLs with three main research categories in mind:

- design recovery (section 1.4.1),
- re-engineering/reverse engineering (section 1.4.2), and
- code translation (section 1.4.3).

The approaches above have been evaluated for their correctness, as discussed in section 1.4.4.

1.4.1 Design Recovery

Design recovery aims to visualise the system using standard models. The extracted models illustrate the system components and show the relationships between them. Techniques used in design recovery do not make changes to the code-base or modify how the system looks. However, the system could be updated from the design using Round-trip design recovery, Albin-Amiot and Guéhéneuc [48], also called Round-trip Engineering (RTE) techniques.

Unified Modeling Language (UML) is a standard language for modelling and depicting the recovered components in third-generation programming languages and object-oriented programming languages, Lavagno et al. [49].

However, following UML or a similar modelling language approach is not viable. This is because, to the best of the author's knowledge, there has not been any modelling language that supports 4GLs fully.

Using UML to illustrate any fourth-generation programming languages is not always practical. UML is designed to support object-oriented design and pictures the static and behavioural model of third-generation systems. In addition, UML is not a formal language, albeit there have been attempts to formalise it.

Fourth-generation programming languages encapsulate routines into keywords. The source code for these languages is closed. Therefore, UML can do a black-box modelling for a system but in-depth modelling requires revealing the language syntactically. The algorithms used in deploying 4GLs functions are not therefore visible within UML.

Researchers such as Wu et al. [50], have asserted that generally, legacy systems are large and complex. Complexity increases the difficulty in recovering the internal design where the design is not contained in the implementation.

1.4.2 Re-engineering

According to Harrison et al. [51], researchers should build a knowledge base in order to feed information into the system engine. The knowledge-base engine uses these findings to depict the relevant model. Therefore, to use the same methodology on 4GL systems, it may be necessary to examine the code in order to simplify it by grouping together lines of code based on functionality, purpose or behaviour.

Re-engineering is another solution that could be followed to renew, fix, or refactor a legacy system. In re-engineering refactoring adds improvements to the underlying infrastructure of a system. This means that re-engineering may compromise the design conformance. Re-engineering applies

changes to the structure and internal components and the links between them. The design change can impact the user requirements which were used to build the legacy system.

Using re-engineering the system could be modernised and legacy components ported onto an up to date platform. Re-engineering does not necessarily demand a change to the set of technologies that was used to build the legacy system. A legacy system can be rewritten to meet the changed business needs with the same programming languages used to build the legacy one.

Re-engineering requires keeping the 4GL in use which means that modernisation is subject to the limitation of the 4GL system previously used. Therefore, replacing the system would require a full transformation using reverse engineering tools and compiler-generator tools.

Decision makers at organisations can be determined to completely transform the legacy system that they use. They in essence have to be prepared to cover the cost involved in discovery old obscure bugs, data structure changes, and new bugs introduced during the transformation process in addition to the costs of disruption to ongoing business. In this case code translation is the method that is commonly used.

1.4.3 Code Translation

Code translation works by parsing the targeted system code line by line in order to interpret the language syntax. Researchers could either translate the code into an intermediate language as an initial step or directly translate it into the desired language. In the case of 4GLs, the only source provided for understanding the language syntax is the documentation provided by the language creator.

In this case, evaluation depends totally on the output behaviour of the re-engineered system. The system output should function and return the same results as the original system. Testing, even if rigorous, cannot find all the discrepancies in a system or between two systems, Kumar and Syed [52]. It follows that proving the behaviour of the reverse engineered system is fully equivalent to the original, is equally challenging.

Software comprehension is a complex process when it comes to disassembling legacy system components into their basic elements. Software comprehension methods help in bringing to the surface the embedded logic, data flows, and architectural structure. System types vary in essence. Object-oriented ones targeted wide audiences from the beginning. However, not all systems are implemented using common technologies or tools.

1.4.4 Reverse Engineering Approaches for Correctness

Once reverse engineering has been attempted there needs to be a robust method to check that the translated language matches the original, in terms of syntax, behaviour or both aspects. The scope of this thesis concentrates purely on the syntactical aspects.

Possible methods for determining the correctness of the results of such translated 4GLs are:

- Screen output,
- Unit testing,
- Chunking trees.

Screen output and unit testing have been used for both the behavioural (semantic) aspects of the translation as well as syntactical aspects, but without enabling any clear distinction between the two.

The use of chunking trees was introduced by the author of this research as a novel method to enable the comparison between two language's syntax.

1.5 4GLs in Industry and the ESIS Case Study

1.5.1 4GLs in Education and Industry

Fourth-generation programming languages (4GLs) have gained popularity over years of development, Lehman and Wetherbe [53]. 4GLs are known to allow for rapid software systems development so minimal training is required to have one team skilled to create substantial applications. For this reason, 4GLs were adopted widely by corporations in the industrial and educational sectors (Chandrasekran and Broadwater [54], Dolado [55], Arnett and Jones [56]).

Fourth-generation programming languages theories and practices were considered in the 1990s to be valid educational material, McCracken et al. [57]. Therefore, educational institutions accepted including them in the syllabus where students could learn employable skills and research these types of languages.

Early in the development of 4GLs, a fierce debate was sparked off. Educational organisations questioned whether 4GLs should be taught to computer scientists or not (Lehman and Wetherbe [53], McCracken et al. [57]). Industry had concerns about the stability, security, support, and upgrade features which are vital to keep their business products up to date and which might be threatened by embodying cutting-edge technologies.

Education, accepted that 4GLs were languages which embraced all aspects of programming languages and logic. They also provided useful employability skills for their students. So universities taught these languages and encouraged researchers to establish studies of them. However, it was not an obvious task for researchers due to the high cost involved. In addition, some software licences prevented engineers from reverse engineering business tools such as 4GLs.

Fourth-generation programming languages were also used to customise large and complex systems, add programmable features to extend the application functionality.

Advanced Business Application Programming (ABAP) (Chawla et al.'s US Patent [58]) was an example of these languages which were built by the multinational enterprise software supplier SAP (Jacobs et al. [59]) for their Enterprise Resource Planning (ERP) system. Developers could query the database, manipulate data in different sections and create new services for front-end developers to consume.

Another example of a 4GL was Structured Query Language (SQL) (Kipp [45], Toyama [60], Monge and Schultz [61]), which enabled the querying of databases using semi-natural language syntax such as "create table", "select from", "sort by". Although different database engines use different syntaxes of SQL, there was a high similarity among the SQL dialects.

Uniface, (Uniface.com [62]) is an example of a 4GL which is also a domain specific language. Uniface can connect to more than one data source at the same time. Even more, it can bind tables of data coming from different sources as if those tables were originally located in the same system. This powerful feature made Uniface competitive in the market. To the best of the author's knowledge many educational platforms were built using Uniface, especially in the United Kingdom.

1.5.2 Case study: University of Essex, ESIS and the Uniface 4GL

1.5.2.1 Electronic Student Information System (ESIS)

The University of Essex has suffered from the high cost of maintaining an internal legacy system. This was a key motivation for the choice to use the University of Essex Electronic Student Information System (ESIS) as the main Uniface case-study for this research. This system manages students' data and meta-data and includes the following:

- information about their accommodation and meta-data such as the accommodation access code,
- yearly modules,
- academic qualifications,
- additional qualifications for international students such as the visa requirements and status and English language assessments,
- graduation ceremonies and other teaching events,
- authorisation and access permissions for the university employees into ESIS
- departmental information such as the processes to grant, students' deferrals and tax exemption,
- bank holidays,
- payments summary and financial meta-data,

- student's publications meta-data,
- admission status.

This list of data is not exhaustive due to the rich functionality and large number of features supported by the ESIS system.

The system's features are changeable since it is under constant development. It is built and maintained internally by the university. ESIS is used by the University of Essex's employees to store and retrieve the information that they need to assist students and enrol them into courses and register new students.

1.5.3 Case Study: Essex University ESIS System

To cope with the illustrated difficulties, different techniques and approaches were considered to explore existing legacy systems. The example chosen as a worked case study was the Electronic Student Information System (ESIS) at the University of Essex. There is a need to define the tools required to approach this 4GL system and to analyse it. This formulated the first step towards understanding the scale of difficulty and to develop a coherent plan to extract the system design.

The University of Essex has built the educational student system using Uniface. The development of this massive application has been ongoing for more than ten years. The author extracted about 6.8 millions lines of code formatted in Extensible Markup Language (XML).

There are often a large number of forms/user interfaces that were built, where each form served different functionality. The extracted XML represents both the graphical user interface and the programming code nested within pre-defined events to handle user requests.

Other research on proprietary 4GLs, which follow a similar behaviour and design pattern could potential benefit from the methodology and techniques demonstrated by this thesis. Such systems are widespread in organisations that require master data management using Customer Relationship Management (CRM) and Enterprise Resource Planning (ERP) platforms.

The back-end system of ESIS handles user requests which are collected from a variety of front-end systems which are installed on the client terminal. Each category of data is named a functional module, where each of them segregates data into a number of fields which mirror the back-end database tables.

1.5.3.1 ESIS and Uniface 4GL

The ESIS system has been built with a 4GL named Uniface. The initial investigation for this thesis, raised concerns about the language syntax of the programming language used to implement ESIS. Uniface characteristics suggest that the language is domain-specific and falls under the category of fourth-generation programming languages (4GLs). Further information about domain-specific

languages and 4GLs are provided in sections 1.2.4, and 1.3 respectively. 4GLs are not considered to be user-friendly and are difficult for maintenance or extensibility, Zhu et al. [63].

The source code of the Uniface has never been publicly provided by the vendor. The aim of this research has not been to translate the original language which was used to build the Uniface compiler and IDE. The aim of this thesis has been to translate/reverse engineer Uniface 4GL syntax. Having access to Uniface source code would have vastly helped this author's research in understanding how Uniface interprets its keywords and internal functions.

Uniface does not support interoperability to build projects by adopting and running an external language in its environment. Figure 1.3 taken from Bissayande et al. [3] shows the top languages that support interoperability where “the thicker a line between two languages, the more GitHub projects that contain code written in these two languages”.

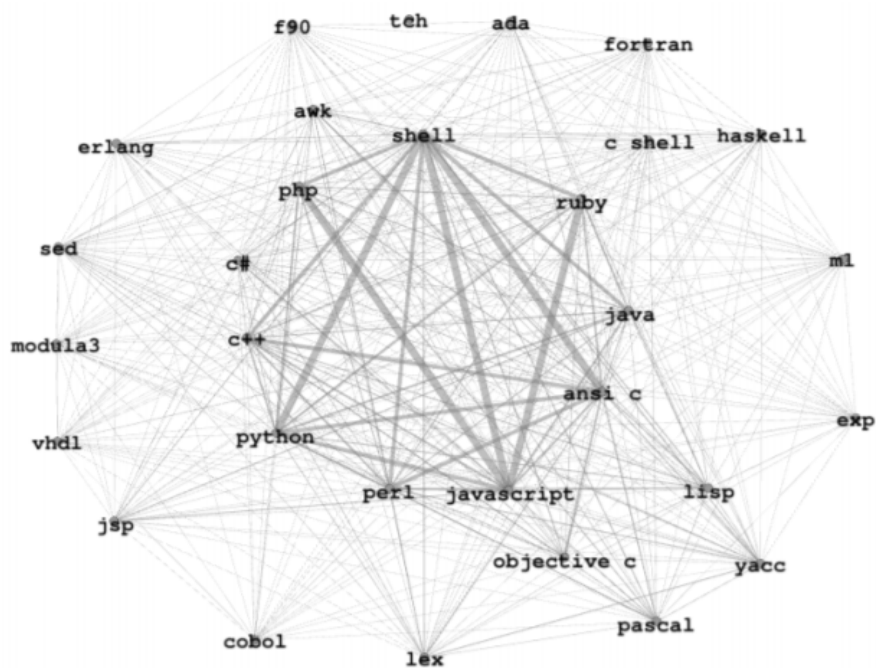

Figure 1.3: Languages Interoperability. (Bissyandé et al. [3])

In an attempt to work around Uniface's limitations and the absence of interoperability, ESIS has been designed to contain lengthy blocks of variables of strings of a dynamic language code such as Python or JavaScript. Performing an operation beyond Uniface's standard capability is done using the dynamic language to export the string variable content into an external file with the appropriate extension, ‘py’ for Python and ‘js’ for JavaScript, and calling the pre-installed language interpreter to run the script file.

This practice of coding is not part of a known software engineering or coding standard and so is fraught with difficulties. One issue is that the string variables are limited in length. Therefore, storing code snippets and exporting them externally may cause data loss. This might lead to silent failure in running the snippets and yield undesirable outcome at run-time in production environments. Uniface parses the dynamic language code variables as strings. Therefore, Uniface does not validate the Python or JavaScript syntax or enable developers to debug the dynamic code at run time since it runs externally. Additionally, it is not possible to determine when the external processes are complete and inspect the returned outcome - whether successful or in an error condition for example.

1.5.3.2 ESIS Data Structures

The student data within ESIS is mainly stored across more than 900 tables in 5 instances of Microsoft SQL Server, comma delimited files, Microsoft Excel files, and text files. In addition, Uniface uses its own meta-data database to hold in persistent form, developers' code, data models, and Uniface specific data. In investigating the meta-data database, the author found that extracting artefacts from the meta-data database was beset with significant challenges.

1.6 Research Approach

1.6.1 Research Direction for this Thesis

Recovering the underlying design of a system built using a proprietary 4GL, in the absence of reliable updated system documentation, is not always feasible.

This research originally aimed at uncovering the design of a Uniface system for a particular case study, using its meta-data database but faced a dead-end. Therefore the author for this thesis followed a novel approach to translate the syntax of the case-study code. In order to show that the output code syntactically matched the original, a novel comparison method was used. These methods were also demonstrated in principle on other proprietary 4GLs and a bespoke 4GL (Mini-4GL).

This thesis investigated to a limited extent some relevant areas that contribute to systems design, formal approaches to model a system. It also studied any possible dissimilarities between a system's design and its implementation (Alzahrani, Yafi et al., 2015 [64], Alzahrani, Yafi et al., 2014a [65], Alzahrani, Yafi et al., 2014b [12]).

1.6.2 Research Aims

The research in this thesis has aimed to provide a novel method

1. To reverse engineer an exemplar model-driven proprietary 4GL syntax (in this case from the Uniface 4GL programming language syntax) into another language syntax (in this into

case C#) in detail but not comprehensively, using a case study (in this case the University of Essex ESIS system),

    (a) To provide proof of concept code to demonstrate the validity of the novel method,

    (b) To reverse engineer just the syntax of the language (excluding the semantics/behavioural aspects),

2. To provide limited exploration across a number of other 4GLs, to demonstrate the same method's broader applicability,

3. To examine a possible methodology, tools, and techniques to find those suitable for such re-engineering.

1.6.3 Research Objectives

The list below set out the objectives of this research.

- Extract the Uniface 4GL source code from the ESIS system.
- Validate the extracted source code of ESIS, that it is text based and can be parsed as a sequence of characters.
- Provide a method to omit non-Uniface specific characters from the extracted code.
- Provide a method to segment the extracted into manageable blocks.
- Create novel grammar rules to parse the extracted code.
- Create novel parsers to parse the extracted code.
- Compare the introduced parser for feasibility and performance.
- Create a novel compiler to translate the parsed code into the target language (C#).
- Verify that the syntax of the input code (Uniface 4GL) correctly matches that of the output code (C#).
- Create a novel 4GL that is similar in its characteristics to Uniface.
- Apply the methods used to reverse engineer Uniface on the novel 4GL and confirm the validity of the syntax of the output.
- Generalise the same method over other 4GLs (Informix and Apex) and confirm it's feasibility.

1.6.4 Contribution to Knowledge and Societal Benefits

1.6.4.1 Contribution to Knowledge

These are set out briefly here, and in chapter 11 (Conclusions) particularly section 11.3 (Summary of Contributions to Knowledge). Each contribution has also been highlighted at relevant points throughout the thesis.

Applying Computer Based Software Engineering (CBSE) Methods to a model-driven proprietary 4GL

This research has applied CBSE methodologies (as explained in section 3.3.2) to reverse engineering a XML-like, model-driven proprietary 4GL such as Uniface. This was novel because up to this research Uniface, as a proprietary language, had not been reverse engineered, and the task of reverse engineering such a language had been approached independently from the methodology used to achieve that task.

Using Encapsulated Document Object Model (EDOM) for processing XML-like code into a list of valid XML files

The research in this thesis used EDOM techniques to segment, fragment and split the large source XML-like code file, into a valid list of XML files.

The novelty was that it allowed the XML-like code containing Uniface programming language structures to be decomposed into schema (DTD or XSD) and parsing streams of XML as chains of characters. These chains included XML characters which had a particular meaning within Uniface.

Novel Grammar Rules for Uniface

As previously described, Uniface was a proprietary programming language, whose full specifications had not been published either by the authors, or in any academic literature. The derivation of grammar rules for a set of its grammar was a novelty and in turn led to a successful method to reverse engineer the parts of that language for which grammar rules had been created by the author.

Use of Parser-Parser and Parser Generators to Translate a XML-like Proprietary 4GL into a 3GL

These techniques had been previously used in the field of natural languages rather than for computer languages - in this study a 4GL.

In this study, the author has applied parser-parser and parser-generators with Left-to-right, Leftmost derivation (LL) and Look-Ahead Left-Right (LALR) parser (LALR) techniques to enable a formal approach to parsing computer languages with XML-like model-driven proprietary 4GLs.

Using Chunking Trees to Syntactically Compare Languages

This study has just been concerned with a novel method to reverse engineer the syntax of a particular class of programming languages (as opposed to its semantics or behaviour), and produce proof of concept code to demonstrate that it worked. In order to prove that the input programming language syntax matched the output target language syntax, a method was devised or comparing and matching their Abstract Syntax Trees (ASTs) for particular language grammar structures. As the ASTs were, in themselves, too complex for such a comparison they were compared in the form of chunking trees, from which such comparisons could be derived.

This enabled the author to demonstrate valid and matching output syntax compared to the input language syntax.

1.6.4.2 Benefits To Society

Computer software has increased in importance and complexity over the last 70 years, as set out in the discussion above:

- Why Reverse Engineering? (section 1.1.1),
- Why Computer Languages? (section 1.1.2).

The difficulties of reverse engineering legacy systems has been discussed in section 1.2.3 (Grappling with Reverse Engineering Legacy Systems).

Organisations attempting to maintain legacy 4GL systems were faced with the invidious choice between:

- continuing to maintain the legacy system at increasing expense, and with a decreasing pool of expertise,
- rewriting them from scratch in a more maintainable language. This could be very costly and time-consuming and had a relatively high risk in the early stages whilst obscure bugs were ironed out,
- reverse engineering the legacy 4GL system into a language that was more maintainable and familiar to today's software engineers.

This thesis has taken a significant step forward in being able to address the third of these options. The problems of reverse engineering have been increased by the proprietary nature of the 4GL studied (e.g. Uniface) meaning that the full specification and syntax has not been readily available.

Being able to re-engineer such 4GLs thus resolves a significant problem for organisations faced with keeping such legacy systems operational.

1.6.5 Technical Outcomes

The technical outcomes of this research consists of

- the extracted Uniface code from the XML-like (appendix B),
- the originating forward engineering code that generated the code in the selected targeted languages C# (appendix A),
- valid segments of Uniface with cleaned and corrected XML (appendix G),
- code sample to transpile the author's Mini-4GL (appendix H),
- the reversed grammar and parsing table (appendix D)

1.7 Research Questions

The main research question is:
What methods can be used to automate the process of translating the code written with a 4GL (such as Uniface as an exemplar) into a different language syntax such as third-generation language syntax, (C# for instance), in the absence of documentation and syntax definition resources?

This leads to two further sub-questions, which this thesis has addressed:

What development methodologies, that support modular programming and employ reverse engineering techniques, can be applied to 4GL code, and in particular proprietary 4GL code such as Uniface?

What parsing techniques may be used to generate a parse table for the syntax of Uniface in particular and other proprietary 4GLs in general, that is described and represented by a set of grammar rules?

1.8 Thesis Structure

Chapter 2 Literature Review presents an overview of 4GLs literature and illustrates the characteristics of these languages together with the methods and tools used to reverse engineer them. The chapter also covers literature on the challenges that researchers and developers may face when aiming to reverse engineer legacy systems involving 4GLs.

Chapter 3 Methodology firstly illustrates the methodology used to investigate the research questions, study Uniface, and investigate the problems that were encountered when diagnosing the symptoms of failures. Secondly this chapter introduces the methodologies that were used for implementation, especially those involving an incremental delivery approach. Thirdly it explains in further detail the methodology employed to support the proof of concept solution and framework design.

In **Chapter 4 Implementation I - Extraction, ER and Relational Algebra Modelling** a solution has been proposed to solve the problems of reverse engineering Uniface. The chapter gives examples used to target 4GLs syntax for formalisation and design recovery. In this chapter the difficulties with each approach are investigated, Uniface-specific hurdles are overcome and solutions such as Encapsulated Document Object Model (EDOM) are proposed.

Chapter 5 Implementation II - Reverse Engineering Uniface Schema firstly illustrates the author's visual representation of Uniface schema code using Document Type Definition (DTD) and XML Schema Definition (XSD). Secondly this chapter offers corrections for obscure Uniface characters that would otherwise cause XML scanners to fail.

Chapter 6 Implementation III - Extracting Uniface Elements firstly this chapter articulates the methods used to extract Uniface internal components based upon the structures set out in chapter 5.

Secondly this chapter also sets out the algorithms used to deliver the author's proposed system.

Chapter 7 Implementation IV - Grammar Formalism describes Uniface syntax formalisation techniques.

Firstly, it covers the topics of modular grammar and grammar engineering which serve the purpose of writing a valid grammar for the Uniface language.

Secondly it delivers a visual illustration of the code blocks using an abstract syntax tree.

Thirdly, it shows the data-flow between the different layers of the proposed system.

Chapter 8 Using Parser Generators to Implement Uniface Grammar firstly introduces the case-study and the frameworks used to implement it.

Secondly it explains the methodologies used to work with these tools and the approaches considered to keep the proposed system aligned with theory.

Thirdly, it then discusses in detail each of the two parser tools used in the implementation phase of the project to generate code (Gold Parser and ANTLR) and also provides a comparison between the two tools.

Chapter 9 Extending from Uniface to Other 4GLs

This chapter covers the implementation, to a very limited extent, of the author's method to derive syntax as part of reverse engineering, for two other 4GLs (Informix and Apex) as well as for Mini-4GL. Mini-4GL was a model-driven SQL-like language of the author's invention. These languages have been used to show the broader application, in principle, that the novel method as applied in more detail to Uniface, can also be used on other 4GLs.

Chapter 10 Results provides a novel methodology for demonstrating results that the author's method has achieved its objective. It shows for the syntax and grammar rules defined in this

proof of concept to the extent as implemented, that the inputs (in Uniface) completely match the syntax of the transformed output (in C#), thus demonstrating that the reverse engineering was successful. It also shows examples where there has been no match, either because no grammar rules have yet been defined, or because of a language-specific construct.

Similar results are shown for the other 4GLs (Informix and Apex) and for the the author's Mini-4GL.

Chapter 11 Conclusions provides a summary of this research and illustrate the novel methods and tools used to deliver the content presented by the author.

The chapter also shows the current and future possible levels of automation and provides some ideas both for improvements to the current proof of concept system and also for future research work in this field, based on the author's novel method.

Chapter 2

Literature Review

This chapter outlines the current state of research and knowledge of key areas in the subject including:

- 4GL Literature (section 2.1),
- Methods Used to Reverse Engineer 4GLs (section 2.2), comprising:
  - Knowledge-Based Software Engineering (KBSE) (subsection 2.2.1),
  - Design Recovery (subsection 2.2.2),
  - Purdue Compiler Construction Tool Set (PCCTS) subsection 2.2.3),
- Supporting Techniques and Concepts (section 2.3), comprising:
  - Code Generation (subsection 2.2.4),
  - Design Patterns (subsection 2.3.1),
  - Evolution of Abstract Syntax Trees (subsection 2.3.2),
- Re-engineering/Reverse Engineering Legacy Systems Techniques (section 2.4)
- Re-engineering/Reverse Engineering Legacy 4GLs - Research Gap (section 2.5).

2.1 4GL Literature

In this section the author has surveyed the relevant academic papers on 4GLs.

According to Zhao [66], programming languages have evolved from being machine-centric to being application-centric. Fourth-generation programming languages are domain specific. Software

corporations designed 4GLs to increase the level of productivity and reusability. This is achieved using branches and sub-branches of Software Engineering such as:

- Feature-Oriented Programming (Batory et al. [67]),
- Generative Programming (Czarnecki, [68]),
- Domain Specific Modelling (Kelly and Tolvanen [69]) and
- Model-Driven Architecture (Evans [70]).

The main features of 4GLs are clearly defined and so researchers are familiar with the nature of those languages.

Ketler and Smith [15] illustrated the significant changes brought in by fourth-generation programming languages and compared the performance of developers who used these languages versus developers who used mainframe languages. Karen et al. classified the productivity performance improvements into two groupings - environmental and personal. In addition, this paper suggested a procedure to match the individual's attributes with the characteristics of environment factors.

A summary of Ketler and Smith's [15] findings is presented in table 2.1:

3GL	**4GL**
Organisational Traits	
Requires large numbers of lines of detailed code	Is less detailed and shorter
Difficult to enhance and maintain	Easier to enhance and maintain
Require less resources	Require more machine time and overhead
Follow community standards	Lack standardisation
Requires structured environment	Environment agnostic
Personal Traits	
Require more programming skills	Allow for rapid development and customisation
Require technically trained person	Require less programming skills
Can be programmed by individuals	Demand more team work
Require precision, attention to details, and concentration	Require less thinking

Table 2.1: Comparison of 3GL and 4GL Traits according to Ketler and Smith [15]

4GLs gained popularity because of their ability to create programs rapidly with relatively less lines of code compared to 3GLs. Ergo, less lines of code requires less efforts to maintain. This is true in the context of companies which can afford the cost of hiring and qualifying new members of the team.

4GLs are reputed for their richness in the number of built-in functions. These functions are used as of the shelf utilities. The source code of these functions is not amendable and therefore, the holder component can be maintained to correct its use of the function but unlike 3GLs, components cannot change their behaviour to meet growing business demands. As a workaround, developers rebuild and replace existing components with ones that better meet the new business need.

4GLs can be maintained to preserve the state of the system but they cannot be made adaptable to embrace new changes. This, in addition to their lack of standardisation and higher cost to run reverted the adoption decision and promoted 3GLs over in many cases including ESIS (the case-study of this research).

Tharp [71] and Allen et al. [72] studied 4GLs from an academic perspective. This study differentiated between third-generation object-oriented programming languages and fourth-generation ones in terms of complexity, syntax, and features. Tharp concluded that 4GLs had, by the mid 1980s, been adopted by commercial industries and academics. This shows the significance of 4GLs and motivates the research in this thesis to target commercial products and to adopt a case study in that class of organisation.

2.2 Methods Used to Reverse Engineer 4GLs

This section discusses existing methods that have been used to reverse engineer 4GLs.

2.2.1 Knowledge-Based Software Engineering (KBSE)

Advances in compiler implementation techniques since the 4GLs were originally designed and implemented, have raised the level of abstraction that is possible for the language's syntax. Such new implementation techniques, which aim to deliver better optimisation could lead to discrepancies between the formal design and semantical implementation of programming languages when executed.

KBSE aims to bridge this gap by introducing a set of formal declarations for the semantics of a program. This of course is quite distinct from the syntactical properties of a programming language.

Lowry [73] in 1993, proposed KBSE methodologies as a semi-automated semantic approach to translate the behaviour of a system based on a formal specifications that could be represented mathematically.

Canfora et al. [4] 2011, built on Lowry's work, and also that of Chikofsky et al. [18]. Canfora et al. proposed storing the formal specification of the targeted system into a structured *information base* as in figure 2.1 (i.e. knowledge-base). Storing this information in the information base enabled queries and analysis during the transformation process.

Canfora et al. [4] as shown in figure 2.1, developed a model building on the core processes as developed by Chikofsky and Cross [18]. (Terms from figure 2.1 have been shown in *italics* to enable the reader to identify them in the figure. The colouring of the figure was adjusted to improve clarity here.)

The software engineer develops artefacts as needed which could include:

- Source code,
- Documentation,
- Test cases,
- Executable(s) and
- Change requests.

These *artefacts* were used to form *software views* which in turn visualised the underlying system structure and provided a means for the software engineer to provide feedback into the *abstractor*.

Later research by Sharma and Gera [74] in 2013, showed that the *Recommendation System* (a complex sub-system in itself), could be used as the principal system to provide feedback.

The *Abstractor* builds views using the data extracted from artefacts in the information base.

The *Analysers* perform the necessary operations to store *abstractors* into a knowledge-base (termed an information base) where analysers can be classified as:

- *Static analysers*: responsible for analysing model specifications (i.e syntax)
- *Dynamic analysers*: responsible for analysing process specifications (i.e. semantics)
- *Hybrid analysers*: are a mix of both static and dynamic analysers.

These three analysers can be found in most KBSE systems.

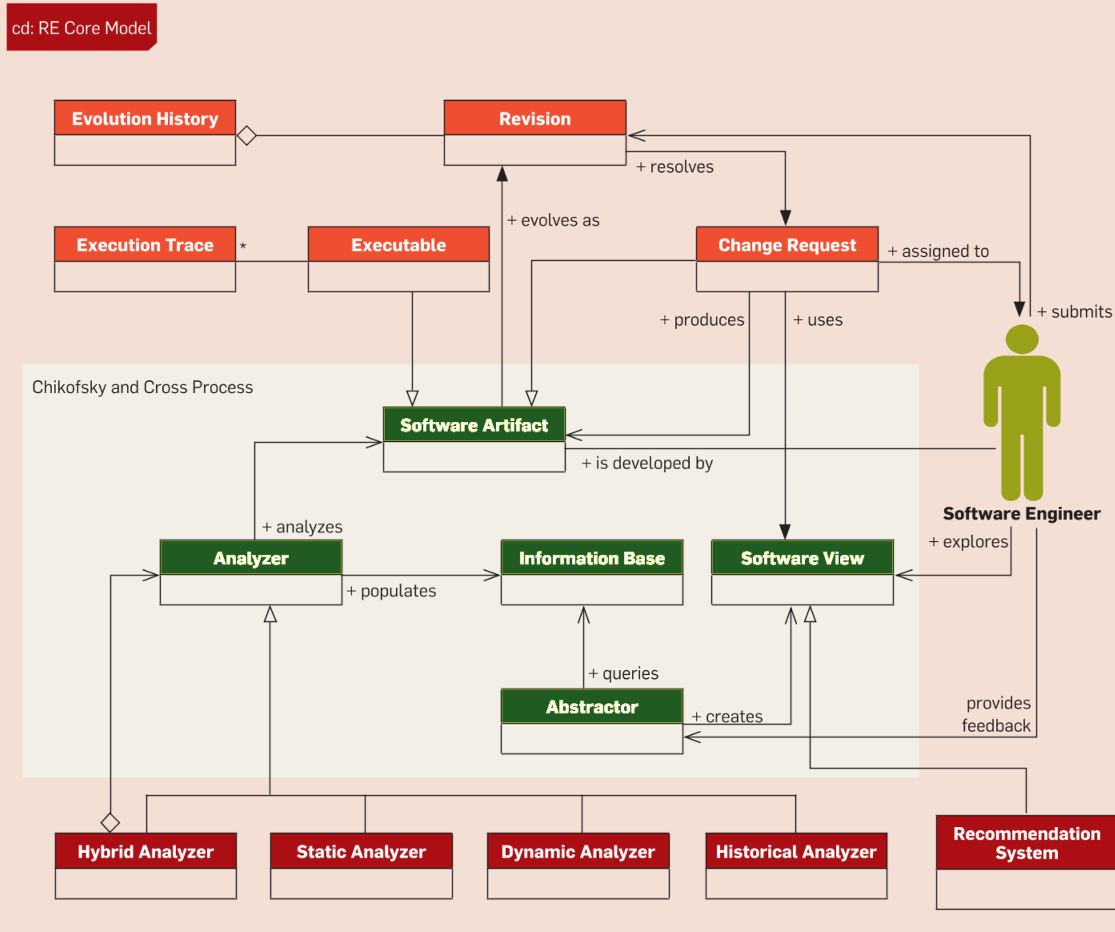

Figure 2.1: Reverse Engineering Using KBSE Concepts - according to Canfora et al. [4]

Harrison et al. [51] have attempted to migrate a 4GL system by constructing a knowledge-base to store the model and required meta-data. Migrating from a 4GL into an object-oriented system can be costly and might require endless effort to yield tightness between the language and a specific database vendor. Harrison et al.'s research was motivated by the inability of 'Ingres ABF (Application By Form)' to work with multiple database vendors. Their research aim was to translate Ingres ABF into Oracle Developer/Designer 2000.

They proposed two approaches to build the tool. The first approach was emulation, where each line of code in the 4GL language was translated into a line or set of lines of the chosen language whilst keeping the function semantically intact. The second approach was to use design recovery techniques.

According to Harrison et al. [51], the former approach (emulation) was the preferred approach because it was more sensitive in recognising output errors, in comparison with the latter technique (design recovery).

The researchers utilised a commercial KBSE database software package (Software Refinery) to store translation rules. The knowledge-base contained a list of rules that provided translation instructions for each of the following structures:

- tables referenced in embedded SQL queries,
- lists of fields extracted from the user interface and source code,
- the relationship between the fields in the database,
- lists of columns referenced in embedded SQL queries,
- the components found on the user interface.

Harrison et al.'s research was limited by the unavailability of the complete source-code.

The researchers used qualitative measurements rather than quantitative metrics to measure the success of their translation process using KBSE.

- **Maintainability:** where the target system should be as maintainable as the source system (although no measure for maintainability was provided),
- **Efficiency:** where the target system should be as efficient as the source. For example, the number of DML (Data Manipulation Language) calls requested by the server should be "approximately similar" to the number of requests by the source language,
- **Completeness:** as measure by the ability to perform the same semantic behavioural functionality as the source system. The authors did not provide any explicit unit of measure for this.

Harrison et al. [51] concluded that

- The target system was *approximately* similar to the source system (without offering further data or units of measurement),
- An emulation approach satisfied both completeness and efficiency criteria but not maintainability,
- A design recovery approach satisfied maintainability and efficiency but not completeness criteria.

The research in this thesis follows on from the emulation technique direction which Harrison et al. [51] suggested to be more comprehensive and promising.

Harrison et al.'s work confirm the difficulty of attempting to derive satisfactory quantitative metrics to measure the success of their translation process. Similarly, Harrison et al. provided no metric for completeness.

2.2.2 Design Recovery

Chikofsky et al. [18] defined design recovery as:

> "the sub-set of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself."

Sartipi et al. [6] proposed using data-mining for design recovery. The researchers used it to partition the legacy system and decompose its modules and functional units based on the design description. The design description was given as an input by the user. The input was written in Architectural Query Language (AQL) that was created by Sartipi et al.

Sartipi et al. investigated the potential for extracting the system artefacts using the frequency of transactions and calls to the underlying database. The authors reported a success rate of 60% in recovering architecture matching the system documentation of which 10% (6% of the total system) was noise. The authors did not provide a method to eliminate the noise.

This thesis's research has not taken Sartipi et al's data mining approach. This was due to

- a focus on the behavioural semantical aspects of the system whereas the author's research has been focused on syntactical aspects, and
- the limitations in retrieving meaningful and complete information from Uniface meta-database 3.1.1.

Chikofsky et al.[18], Harrison and Berglas [19], Sadiq and Waheed [20], and Sartipi et al. [6] each proposed to rediscover the design information and import it into tools that would auto generate the targeted system using the preferred language.

Because 4GLs strive to cover all aspects of a specific domain, they struggle to support cutting edge technologies which evolved outside the 4GL focus. Although there have been tools that target third-generation programming languages the support for manipulating 4GLs has been limited.

Nagy et al. (2011) [75] ported a medical application onto a modern platform, the old system having used the *Magic 4GL* programming language. The authors claimed to have used design recovery to reveal the system architecture and the relationship between the system components and database. In order to achieve this goal they created a tool that operated on Magic syntax to reverse engineer it and carry out a set of forward engineering tasks. They adopted and

manipulated the Columbus Reverse Engineering technique, Ferenc et al. [76], which had been originally aimed at object-oriented applications and used it to understand the relationship between rapidly evolving software components.

Nagy et al. aim was to define a methodology that extracted high level information which illustrated the design recovery data in views. These views reflected the architectural structure, semantics and the relations among databases objects of the system. According to Nagy et al. (2010) [77] it was necessary to have an abstract understanding of the original code before manipulating it. Therefore, coherent and advanced methodologies could potentially contribute to the quality of the re-engineered system.

Nagy et al. managed to extract the source code of the Magic application and the user interface data. The researchers parsed the source code using an existing parser that was capable of parsing Magic-specific ‘Abstract Semantic Graph’. These graphs defined the internal architecture of the components in the application and the interrelationships between them.

Nagy et al. (2010) [77] concluded that research in the field of re-engineering 4GLs is underpopulated and therefore, further research is needed to address the complexities in this domain.

This thesis's research provides a new method to reverse engineer a class of 4GLs, by working with 4GLs that do not have a parser that comes pre-configured for that language, unlike Magic for which such a suitable parser was available.

Yamamoto et al. [78] considered 4GLs as part of a means of establishing a platform for distributed systems. Their platform, called Visual and General User Interface for Databases (VGUIDE), provided a visual tool to run full-scale information systems between the client and several hosts. The authors claim that the 4GL system which was behaving as a middleware layer offered the flexibility to port on to diverse platforms and possessed the scalability to adapt to Online Transaction Processing (OLTP) for heavily loaded systems.

The research in this thesis was not intend to target distributed systems. The technique used in the Yamamoto et al. paper is relevant for concurrent processes, which are irrelevant for the purposes of the research in this thesis.

According to Harrison et al. (1995) [79], reverse engineering or attempting a translation of a system from source-language to another source language (source-to-source) is beset with difficulties. Harrison et al. (1995) describe this process as:

> “Extreme Difficulty: Source-to-source conversion aims to generate code in a ‘Target’ language that corresponds to input code in an ‘Origin’ language ...”

Harrison et al. (1995) conducted a sponsored research project [79] and identified design recovery as an applicable approach to migrating 4GLs. In this approach, researchers, were unable to recover the design fully. This is due to the complexity involved in grouping the disparate 4GL components which perform similar functions. Therefore, Harrison et al. (1995), recovered the

designs of fragments and stored design information into a Computer Aided Information Systems Engineering (CAISE) tool. The tool is further described in Beynon-Davies [80].

Recovering the design of a source system written in a 4GL, according to Harrison et al. (1995), required recovering the following information from the source system.

- **DDL Data Definition Language:** The instruction used to manipulate the database objects.
- **SQL queries:** This would cover not only queries which the 4GL system sends to DBMS (Database Management System) but also covers custom queries written in source code using the 4GL system.
- **Encoded internal representation of the user interface:** This is required to extract the elements which users see on the user interface and their X and Y positions.
- **Module network:** The representation of the module/sub-module hierarchy and the arguments passed across modules.

In the research by Harrison et al. (1995), the recovered design was assessed manually by assigning a heuristic reliability factor, which was used as a metric of success.

Harrison et al.'s methods would not be appropriate for the research in this thesis because for the Uniface case study, the author did not have access to the DDL, the SQL or the module network, which all were essential to recover the design according to Harrison et al.

The Harrison et al. study also demonstrates clearly the difficulty in finding a suitable metric of success.

2.2.3 Purdue Compiler Construction Tool Set (PCCTS)

In developing the Purdue Compiler Construction Tool Set (PCCTS), Parr et al., 1992, [81], used open-sourced components from other compiler tools such as YACC Johnson et al. [82], ANTLR Parr and Quong, 1995 [83], and DLG (DFA-based Lexical analyser Generator). PCCTS was designed to offer a better user experience for constructing a compiler and to report errors during the lexical and parser analysis phases.

PCCTS offers easier syntax to build grammar rules, when compared with ANTLR. However, it suffered from the following issues as compared to ANTLR.

- PCCTS was not well documented and supported the C language only. In contrast, the ANTLR community has provided a wide range of grammar rules for a variety of known programming languages.
- PCCTS grammar rules could be ambiguous, which was not the case for ANTLR.

An example of ambiguous code is given in the code snippet below:

```
1      a :
2          A B
3          |
4          A C ;
```

```
Line 1  a:       defines a grammar rule with the title a
Line 2     A B   sets out the rule that B must follow A
Line 3     |     OR
Line 4     A C   sets out the rule that C must follow A
Line 4         ; ends the grammar rule.
```

In this snippet there is a choice (and therefore a potential ambiguity) as to whether B or C is permitted to follow A. PCCTS could not handle this, whereas ANTLR could manage to eliminate the ambiguity by using look-ahead LL(K) with $K > 1$.

The research in this thesis has aimed to eliminate the ambiguity at the level of static analysis of grammar rules, so therefore ANTLR was preferred to PCCTS.

2.2.4 Code Generation

Code generation is the process that takes an input (of some form) and uses it to generate valid code in the language of interest according to Kelly and Tolvanen [69]. Code generation is commonly used by compilers. Compilers in principle accept an intermediary representation (byte code) of the parsed input syntax and generate native machine code as executable programs. Muchnick et al. [84] explained that programming languages provide a higher level of abstraction from the hardware. However, the lines of code written in any language must translate to machine code to run. Compilers take the input code and generate machine code according to the particular language's syntactic and semantic rules.

Kelly and Tolvanen [69] looked at Domain Specific Languages which used code generation to either generate machine code or Intermediate Language (IL) code. This enabled it to be either run directly on the machine or handed over to an interpreter which regenerates the machine code from the IL.

Aho el al. [85] proposed using 'Twig', a tree-manipulation language, to construct code generators. They claim that 'Twig' enhances the efficiency of code generation by combining a fast top-down tree-pattern matching algorithm with dynamic programming.

2.3 Supporting Techniques and Concepts

This section covers techniques and concepts which are relevant to the author's approach to reverse engineering 4GLs, but have much wider uses and applicability.

2.3.1 Design Patterns

Dong [86] defines design patterns as designs which document common solutions to recurring problems. Design recovery can be based upon the detection of existing design patterns in the code. Recovering design patterns can expose the design architecture and provide an abstract overview of how a system was constructed and modules interlinked. Design patterns are commonly classified into structural design patterns and behavioural design patterns.

Design patterns have been often depicted using UML diagrams (class diagrams for static architecture and sequence diagrams for behavioural). Because UML is not a formal design language, different developers might introduce implementation variances to the same design pattern. Researchers found methods and techniques to reverse engineer existing systems that implement design patterns and recovered them from the source code.

Budinsky et al. [87], established that design patterns could provide a systematic and standard paradigm to share the expertise knowledge with team members of all levels of experience.

> "Expertise is an intangible but unquestionably valuable commodity."

Budinsky et al. [87] developed a tool that combined the concepts of code generation and design patterns to automatically generate application specific design patterns code based on user input. Their tool was capable of generating object-oriented code using a template-based concept. The tool aimed to provide uniform generated code for developers. It was applicable to a fixed number of applications where the generated code could be embedded into a standalone decoupled component. The tools was a static code generator and used template meta-programming (described in section 7.1.8) to enable embedded variables introduced into the output code.

Keller et al. [88] described the use of Design Pattern Detection (DPD) for reverse engineering, claiming that other techniques neglected the rationale behind design decisions, especially for large scale systems. The authors used a SPOOL environment as described in Schauer et al. [89] for pattern based reverse engineering and concluded that their tool demonstrated a good understanding of the recovered pattern.

De Lucia et al. [90] targeted a subset of design patterns called structural design patterns. The authors proposed using a two-step process to visually represent the recovered patterns. In the first step, the code was parsed to exploit a technique that could be used for visual language recognition. The code was analysed in the second step to select the potential pattern candidates. Their method was limited to object-oriented systems and structural design patterns but was incapable of detecting evolved design patterns.

Since their method was limited to object-oriented systems it was inappropriate for using on 4GL systems.

Rasool et al. (2010) [91] defined sets of design pattern features and applied custom rules, annotations, regular expressions, and database queries to match the defined features in particular

design patterns with their representation in the targeted system's source code.

Rasool et al. claimed that recovering design patterns could be subjective in the absence of a standard and formal implementation of each pattern. This results in design patterns implementation being scatter throughout the source-code, with a lack of rules for detecting their presence. This means that detecting design patterns differs from language to language, and is complicated where some languages encapsulate design patterns within libraries which hinders the process of detection.

Antoniol et al. [92] proposed multi-stage filtering for detecting design patterns. This approach converted both design and code into an intermediate representation and then performed pattern matching on the intermediate representation. That approach was also limited to object-oriented languages and to systems where their design was available. Neither of these conditions were applicable for the research in this thesis.

Design patterns assist developers and software architectures to solve problems that were seen challenging in past. However, design patterns are not formally designed and do not implement equally across different languages. Therefore, recovering design patterns and using available tools that perform design patterns recovery has not yet proved to be a reliable process.

2.3.2 Evolution of Abstract Syntax Trees

The topics discussed in the following sections discuss some concepts used in syntactical language processing both for natural languages and (as in this thesis) for programming languages.

The research in this thesis studied the syntactical features of 4GL proprietary languages using Uniface as an exemplar. It used abstract syntax trees to syntactically decompose the source language.

2.3.2.1 Phrase Structure Grammar (PSG)

Floyd [5] studied the syntax of programming languages using Phrase Structure Grammar (PSG). This was a technique used on natural languages to present an extract of text in a tree-structured form.

For example, the sentence *"John loves Mary"* contains two nouns and a verb. This sentence can be represented by grammar rules that dictate a valid order for words (i.e. the verb follows a noun). The grammar rules for this sentence would be:

```
(sentence) -> (noun) (predicate)
(predicate) -> (verb (noun)
(noun) -> John | Mary
(verb) -> loves.
```

The rules above can be represented by a tree of *Terminals* and *Non-Terminals*. The terminals are found to the right of the sign '->' ('John', 'Mary', and 'loves'). Non-terminals are found to the left of the sign '->'.

The tree that represents this sentence is shown in figure 2.2:

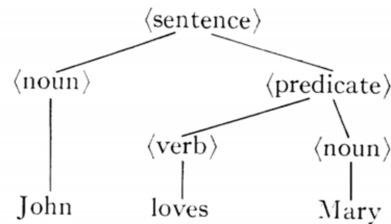

Figure 2.2: Phrase Structure Grammar Tree - Floyd [5]

Researchers in software engineering field have adapted syntactical techniques from natural language processing to process the syntax of programming languages.

Shepherd et al. [93] conducted a case-study to combine the results yielded by an analysis tool together with natural language techniques to analyse method names, class names, and comments.

According to Shepherd et al. reverse engineering tools can provide a comprehensive analysis of a legacy system. However, the output of these tools is very technical and may lack the abstraction and readability that non-technical users need.

Nilsson et al. [94] applied natural language techniques on programming languages to increase the robustness by revealing the abstract syntax tree and parsing it. This approach risked a lack of completeness in terms of accuracy when analysing or generating the tree but it guaranteed not to fail even when the input was incorrect. The approach did not define the grammar rules of the subject languages. The researchers claimed that data-driven parser methods which are created for natural languages are accurate.

2.3.2.2 Parse Tree

A parse tree is a formal extension to phrase structure grammar. Parse trees have been designed to formally represent a program's grammar in a tree structure. Researchers such as Cattell et al. [95], Elson and Rake [96], Glanville and Graham [97] and Cattell [98] all utilised parse trees to represent a valid syntax of the target programming language.

2.3.2.3 Abstract Syntax Tree (AST)

Abstract Syntax Trees (ASTs) are based on parse trees. ASTs for programming languages, represent just the syntactical structure of an application. Therefore ASTs are more suited than parse trees for automated processes which require code syntax input, such as compilers.

Therefore, advanced applications that are capable of detecting code clones Cui et al. [99], translate source code from a language into another Glanville and Graham [97], or tools that provide static code analysis are designed to work with AST Veeramani et al. [100], Neamtiu et al. [101], Rabinovich et al. [102].

The research for this thesis adopted an AST approach derived from novel formal grammar rules to overcome any input ambiguity.

2.4 Re-engineering/Reverse Engineering Legacy Systems Techniques

Re-engineering 4GLs demands solid knowledge of the re-engineering field. In the Guide to the Software Engineering Body of Knowledge (SWEBOK), Bourque and Fairley (eds.) [103], the term re-engineering has been defined as:

> "The examination and alteration of software to reconstitute it in a new form, including the subsequent implementation of the new form".

Researchers have examined the technical challenges in this field and invented techniques and solutions for re-engineering, as set out below.

2.4.1 Re-engineering - Surveys

Chikofsky et al. 1990 [18] maintain that the key objective of software reverse engineering was to improve software maintainability and to enhance the existing systems in terms of performance and functionality. This work defined re-engineering as:

> "...both renovation and reclamation, [it] is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form".

It summarised the key aspects of the need for Computer-Aided Software Engineering (CASE) to assist in computer developments, software evolution, and productivity escalation.

Bellay and Gall 1998 [104] conducted a comprehensive survey of reverse engineering tools. They applied some of those tools to real applications. They analysed the differences, purposes and capability of each tool.

Bellay and Gall reviewed the tools 'Refine/C', 'Imagix 4D', 'Rigi' and 'SNiFF+'. The researchers concluded that these tools would be unable to reverse engineer languages without existing parsers having specific facilities provided for each language. Therefore, utilising these tools for this author's proprietary 4GL research was not feasible.

Canfora and Penta, 2007 [105] surveyed a variety of domain applications where reverse engineering tools and techniques were used. In addition, it used a selection of these reverse engineering tools and presented case studies of their use. The paper categorised the tools into automatic and semi-automatic ones. It also attempted to forecast the future direction of reverse engineering. It encouraged the adoption of reverse engineering tools based on the hypothesis that reverse engineering tools could add value to forward software development practices.

Sadiq and Waheed [20] surveyed different techniques to extract the design from code. This was to better understand the software components and architecture and to tackle the challenges of software modernisation. The objective of their research was not only to completely understand how the system components were clustered but also to discover the relationships between them. The researchers claimed that reverse engineering has gained credibility in business and research because of the complexity involved in maintaining legacy systems, especially in the absence of documentation.

2.4.2 Re-engineering Existing Code to Enhance and Promote Code Reuse

The ‘reusability index’ is a unit of measurement that has been used to asses the reusability of software applications or individual components, Mpatzoglou et al. [106].

According to Bettini et al. [107], large Java applications have been described as having a low index of reusability. This was attributed by the authors to the Java single class inheritance specification. The single inheritance property led to a high level of code duplication in the projects they studied.

Bettini et al. utilised Java hierarchies to overcome the single inheritance feature, using techniques derived from the Smalltalk programming language. These techniques were labelled by Bettini et al. as traits. They defined a trait as an integration mechanism of a class using composition. This approach not only enhanced the refactorization of Java hierarchies but also improved code reuse.

Schwanke [108] targeted modular systems to enhance a code hierarchy by grouping relevant functions into logical modules. The research also discovered a method to identify functions which could be located in the wrong module.

Schwanke used artificial intelligence techniques to create heuristic modularisation advice to improve code quality. Firstly experiments were used to analyse and single out procedures which had been incorrectly placed. Secondly, the method clustered functions according to their relationships. Because this tool used a training system, the author claimed that it would be useful for helping programmers judge similarity between procedures within the same module.

Sommerville's book ‘Software Engineering’ [109] explains the links between software re-usability and dependability. In addition, the book covers the methods used in reverse engineering, code

translation and re-engineering. The author argued that object-oriented programming promotes class with very specific purpose. Therefore, reusing classes require detailed understanding of their purpose and implementation. The desired level of abstraction is not achievable using object-orient philosophy but using components that hold the correct level of abstract implementation.

Sommerville proposed using component-bases software engineering (CBSE) to build independent components. CBSE improved the index of reusability and maintainability of components. Independent components used interfaces to communicate with each other to decouple them whilst maintaining their cohesion.

The research in this thesis adopted the CBSE methodology and provided further details in chapter 3.

Kienle's PhD Thesis in 2006 [110] aimed to provide a solution for reverse engineering researchers to construct reusable tools from off-the-shelf components. The researcher found that building proof of concept tools to support research would be costly. Kienle used techniques using pre-built components and integrated development environments to add more components or to customise existing ones. This method was intended to boost the ability to re-use them in different applications and scenarios.

2.4.3 Re-engineering - Code Analysis and Transformation Tools

Compilers generate machine code application from the input language syntax. Compilers also 'tweak' the code (also termed 'transform' or 'change' the code) according to Binkley [111]. Code transformation can be essential to optimise its performance. Nevertheless, compilers analyse the source code for this purpose where the analysis outcome could be stored in an intermediary form or structure. Code analysis challenged researchers for thirty years according to Binkley. A parser which has been built to match the targeted languages is a prerequisite for language analysis.

Müller, et al. [112] presented the case when the technology used in building the legacy system become out-of-date and therefore, the complexity increases with the need to port the legacy system onto a modern platform. Transformation tools were in more demand when the focus was shifted toward building web technologies rather than OS-specific desktop applications. This was not only challenging because of a dichotomy in programming languages and frameworks that build desktop versus web applications but also due to the dichotomy in technology and resources that power and run web applications.

Transformation tools therefore needed not just to transform the source-code but also needed to be flexible enough to accommodate architecture differences. In this case code analysis and code coverage would not reflect the true percentage of transformed system. The output system may contain far more or far less lines of code or more or less components and still be considered an equivalent to the input system. Estimations on the total transformation process need to be carried out manually using observation and experts judgements, according to Müller et al.

Transformation tools also can provide a mapping method to map components in the source system against transformed target system could be used as a tracking system but without deep analysis on the source system, the progress would remain subjective. The analysis phase is seen a intensive mini-project that precedes the core transformation phase. Therefore, time and budget for it should be made available in advance. In practice, researchers and developers should aim to target a selected number of components/modules/packages to transform the source system based on a subject matter expert's recommendations. Subject matter experts can elect the parts of the system which will used in the long term and can eliminate the legacy parts.

Müller et al. provided a roadmap for researchers to investigate software understanding, comprising software evolution, reverse engineering, automated computer aided tools, database reverse engineering, and tools that help to improve usability. The authors compared a collection of tools favoured both by research and also by industry for reverse engineering. They concluded that ease of use was the main reason for industry to adopt some of these tools.

Eltantawi and Maresca [113] aimed to target code analysis for reverse engineering using logic programming (LP). The researchers used LP to better support multilingual documents. A parser built for language ‘K’ can be extended to support Language ‘P’ in LP with less effort in comparison with a non-LP parser, according to Eltantawi and Maresca.

Eltantawi and Maresca suggested that logic programming separated analysis from the programming code. In their study they showed how to use a logic programming language such as Prolog for code analysis of Clipper (one of the xBase 4GL languages).

Eltantawi and Maresca's approach could handle programming languages with known syntax, that is, grammar rules. Proprietary languages were outside the scope of their study.

2.4.4 Re-engineering - Cost and Complexity

Akers et al. [114] raised the following challenges when dealing with code transformation for large-base systems, as summarised in the following list

- The high cost of running a team to perform the transformation.
- The cost would also be high for commercial transformation systems such as ‘DMS’ (which was used by the research authors to perform C++ transformation). Commercial parser generators such as DMS have been designed for a pre-defined set of programming languages. Extending the product to cover new languages was not an option, since DMS was developed over a period of nine years, so patching the system to overcome it's limitations was not considered feasible.
- The existence of advanced transformation tools such as the ones used by the authors (DMS, and Boeing Bold Stroke) to transform C++ avionics systems did not eliminate the need for manual intervention and expertise.

- The proprietary nature of the source-code licensing limited the number of teams or systems which could have access to the source-code.
- Manual conversion and adjustments had drawbacks on cost, project direction, and output system design.

According to Akers et al.

> "pre-processors, conditionals, templates, and macros can lead to an explosion of possible semantic interpretations of system code and a resource problem for a migration tool".

Therefore, Akers et al. strongly recommend that code syntax manipulation should be preserved as AST.

The research in this thesis has built on the lessons learned from Akers et al. and consequently has used ASTs for syntax transformation.

2.4.5 Re-engineering - Querying the Extracted Knowledge

Sartipi et al. [6] proposed a first phase to perpetuate the knowledge extracted from a legacy system into a database. The second phase would be to scan and parse it using custom user's queries using Architectural Query Language (AQL) as input. The third phase used a modified Branch-and-Bound (B&B) algorithm, Lawler and Wood [115], to maximise matching results for the user query against the knowledge in the database.

Kullbach and Winter [116], suggested that various re-engineering techniques are built over querying layer(s) as shown in figure 2.3. Examples and evaluation were presented in the context of the Graph Repository Query Language (GReQL). The paper revealed the dichotomy between software understanding and renovation. Software understanding comprised of code retrieval, browsing, or measuring whereas renovation consisted of redocumentation, restructuring and remodularisation, according to Kullbach and Winter.

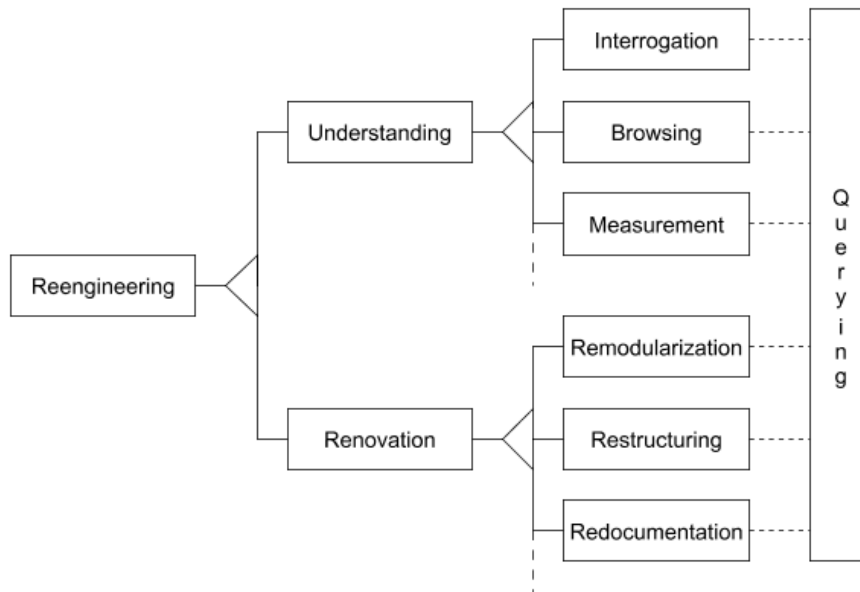

Figure 2.3: Using Querying as the Gate Layer to Perform Various Re-engineering Activities - Sartipi et al. [6]

Ebert et al. [117] in 2008, used the TGraph directed graph (a part of graph theory) to model and query software artefacts using common representation such as XML. The authors claimed that this approach enhanced the flexibility in terms of visualising the system sub-components and translating them into code. Extracting an abstract view of the system, TGraph might facilitate creation of a versatile model for reverse engineering at the analysis phase. In addition, the adoption of Graph Repository Query Language (GReQL) maximises this technique for practical applications.

2.4.6 Re-engineering the Underlying Syntax

Van Den Brand et al. [118], claimed that re-engineering tools originated both from generic programming and also compiler construction fields. They found that it was challenging to formally define syntax and semantics for legacy programming languages.

In the context of re-engineering, Van Den Brand et al. grouped generic programming languages tools into the following classifications:

- **language-independent**: The tool is language agnostic. For instance, tools that perform regular expression search on input text. These tools are independent from the input and can be reused to perform regular expression find/replace on any input.
- **language-dependent**: The tool is dependent on the language which was used. If the tool

is dependent on or configurable for a particular language syntax and language grammar rules, then the tool will only work with that particular language.

- **language-parameterised**: The tool can be dependent on multiple languages but performs on one language at a time. therefore, the language is passed as an input to the tool.

In addition, researchers, Van Den Brand et al. raised research challenges which were observed when applying compiler construction techniques in the re-engineering field. These challenges, without any conclusions provided, have been set out below:

- The complexity required for code analysis is significantly high as set out in section 2.4.3.
- Parser conflicts are expected with parsers that parse multiple languages dialects.
- Data-flow analysis issues such as detecting dependencies of variables, external sources and libraries can be problematic as some of these can only be detected at run-time.
- The analysis phase is a complex process in itself and also visualising it's outcomes is equally as challenging.
- Knowledge representation and knowledge querying is not a straightforward task. This has been discussed further in section 2.4.5.
- Systematic conversion of procedural programs to object-oriented programs and querying programs syntactic information has been under-researched.

2.4.7 Re-engineering Technology Dependent Tools

Perin [119] defined heterogeneous as the systems that were implemented with two or more programming languages. By contrast, homogeneous applications were those implemented using a sole language. Therefore, the extracted XML-like source of Uniface is by definition a heterogeneous process. The extraction contained both XML-like and the developer's code that was written using Uniface language. In addition, other language's source code, such as Python and Windows Shell code, were found in the extraction. These had been used to achieve tasks beyond Uniface's capabilities.

Perin questioned the methods that could be used to reverse engineer heterogeneous applications. The research started from the position that applications adopted a number of technologies to meet their functional requirements. MOOSE (Multiphysics Object-Oriented Simulation Environment) Ducasse et al. [120], was chosen to model the system and analyse dependencies. Perin's research aimed at refining the components' dependency with support for multi-level of abstraction.

Perin's study was dependent on existing tools that could parse the languages which were used in their study. Such tools were not available to use with Uniface.

Dominik et al. [121] identified a key problem in the development particularly of software for

mobile systems. This was a possible loss of data as a consequence of the lack of understanding of the mobile application development life cycle. Even senior developers could be hindered by this. Reverse engineering could add value to their work by spotting bugs and anomalies. There were also cases where there was a mismatch between a software implementation and it's relevant documentation, if any existed. This paper stressed how critical some mobile applications could be, so understanding the mobile application life cycle and ensuring that the intended design matches the implementation would be a crucial prerequisite.

2.4.8 Re-engineering for Reconstructing Design and Documentation

Byrne [122] explained the disadvantageous that arose with the following approaches when reverse-engineer/re-engineer a legacy system:

- **Manually rewriting the system**: This approach requires rewriting new tools and this could be an expensive approach, retaining the undesired structure of the legacy system. Manual translation is a slow process and requires a large and therefore expensive team.
- **Automatic translation**: This approach overcomes the issues with manual translation but could be challenging if the target language does not fully and correctly implement both the syntactical and also the semantic features of the source language. This approach does not facilitate any changes in design structure if these might be required in the target language.
- **Redesign and re-implement the system**: This approach starts with the requirements of the source system to build a new system without the need to investigate the legacy source code. The requirements of the original system must exist in some form. Alternatively, if the current design no longer exists, extracting the requirements from the source-code could be as costly as writing the new system from scratch.

Byrne's [122] method as described in the researcher's case study, was largely a manual one. The steps Byrne followed have been summarised as:

- **Information collection**: In this step, Byrne gathered as much information as possible from the source language (Fortran in this case) from available documentation, source code and expertise.
- **Information examination**: Byrne examined and reviewed the extracted information from the previous step. Examination included breaking down the system into components small enough to provide a high level of abstraction but detailed enough to capture the system's intended behaviour.
- **Functionality recording**: This step recorded the functionality of components in plain English and documented how the processes worked both in terms of business rules and also in terms of business logic.

- **Data flow and control flow**: In this step, the functions were analysed for data and logic flows through the source code. The results of this step were illustrated in flow charts.
- **Review the results**: The final step was reviewing the results from the above steps and rewriting the system's documentation.

The output from Byrne's steps was the production of a detailed set of documentation which could then be used to re-engineer the system and write a new system from scratch.

Müller et al. [123] suggested that more resources should be devoted to assisting industry with software understanding and program analysis. The main focus of their research was on constructing the architectural design of the system. Consequently, as a part of their project, the authors developed a tool (Rigi) to extract an abstract view of the targeted system.

2.4.9 Re-engineering for Recovering the Entity Relationship Modelling

Chen [124] covered Entity Relationship Modelling (ERM), highlighted the merits of the different existing Entity Models and classified them by entity definition, associations, and attributes. The author also advised on model translation, from one type to another.

Castro et al. [125] were Microsoft researchers who had also contributed to this area of computer science.

The ADO.NET Entity Data Modelling (EDM) system contained the Entity Framework which was a set of tools. These had been widely used by developers to implement data interfaces, where the targeted repository was SQL Server. The tool-set included an object-to-relationship mapper which has been enhanced with a visualiser that was integrated within the Microsoft integrated development environment.

The focus of Castro et al.'s research was to help developers to optimise communicate with a central repository.

Adya et al. [126] also used ADO.NET in their paper, to investigate data abstraction. Their research focused on data mapping and other powerful features that have been embodied within the ADO.NET framework Entity Data Modelling (EDM). The compiler was used to generate valid SQL queries that communicated with the connected database. The compiler was designed to be either invoked by the designer (development run-time) or in the form of application run-time. The compiler contained a parser that generated trees that represented the query. These trees were named ‘Canonical Command Trees’.

Research by O'Neil [127] covered Object Relational Mapping (ORM) which mapped object-oriented objects onto database objects. O'Neil [127] used the Hibernate platform for .NET to explore ORM. The technique enabled the connection of multiple applications to one central database.

Reverse engineering legacy systems to maintain the ER design is required to avoid information loss during transition. However, it has been generally more common to design ERs using UML, which is a non-formal language, according to Alzahrani, Yafi et al. [12].

2.4.10 Re-engineering Using Parsers and Parser Generation Technologies

As the aim of this thesis is to target the syntax of 4GL languages, the literature on parsers, parser generation technologies. Theories such as language syntax and parsing expression grammar are also relevant and discussed below.

Warth and Piumarta [128] designed a general purpose tool for generating and manipulating tokenisers, parsers, visitors design pattern, and tree transformers.

The resultant tool, OMeta, was an object-oriented programming library for pattern matching. This tool was described by Warth and Piumarta as useful for experimenting with new programming languages. Therefore, it would be unlikely to be suitable for building a parser for complex systems.

Medeiros et al. [129] formalised regular expressions by transforming them into Parsing Expression Grammars (PEG) using algebraic notations. This technique was useful for languages that could be parsed using regular expression parsers. Due to its limitations in parsing large steams of input, regular expression parsers were less commonly used.

LL(*) parsing strategy was introduced by Parr and Fisher [130] in 2011 with ANTLR expressions used for constructing parsing decisions from ANTLR grammar. This paper introduced a new LL(*) technique to reduce parsing run-time errors and which provided better support for error handling and debugging. The authors did not include strategies to use the parser in contracting or recovering the syntax for legacy systems.

Koprowski and Binsztok [131] aimed to overcome the drawbacks within previous implementations of context free grammar parser generators, by claiming to have a method for a total correctness guaranteed approach. This parser was at an early development stage and a complete proof of concept was yet pending. Therefore, although it could be used with languages that had explicit and well defined syntax grammar rules that method was not sufficiently developed to be used to reverse engineer existing systems.

Packrat parsing algorithm is a type of Parsing Expression Grammar (PEG) algorithm, a type which are designed for performance where the algorithms performs in linear time, Ford [132]. Packrat uses memoization which is a technique used in the field of ‘dynamic programming’ to cache subsequent decisions. Caching can reduce the processing time significantly but it increases the memory usage. The algorithms used in this research have been evaluated in terms of complexity in section 8.6.

Tratt [133] evaluated Packrat parsing in his research and concluded that this method could

produce inaccurate parsers and then suggests an alteration to the original method to improve its accuracy.

Moonen [134] showed how parsers can be written, subject to the existence of a grammar. However, reverse engineering grammars is beset with difficulties. When automating the phase of extracting artefacts, the tool should account for incomplete code or syntax errors.

Saeidi et al. [135] utilised model-driven engineering to build a framework for analysing legacy systems and creating useful tools to assist in the reverse engineering phase. The researchers worked with languages that had known syntactic grammar rules and therefore it was not possible to adopt their tools for this thesis.

2.5 Re-engineering/Reverse Engineering Legacy 4GLs - Research Gap

The research into using parser generator and compiler-compiler tools has been shown to be a promising area, but has not yet solved the challenge of reverse engineering proprietary languages that are shipped without complete syntax grammar rules. Therefore, the research in this thesis aimed at utilising parser generators to provide a new method to work with proprietary 4GLs that have not been visited before for re-engineering/reverse engineering.

Other researchers attempted to reverse engineer proprietary 4GL languages, where they had been able to use existing tools to parse the source language. This approach had so far not existed for Uniface (the principal target for this research).

In summary, the challenges of reverse engineering 4GL systems have been shown above as being non-trivial. The obsolescence and skills gap of many third-generation programming languages and 4GLs, has posed problems. These have encouraged the development community to invest in re-engineering and automating the translation of those languages. However, although 4GLs had gained popularity, due to the level of complexity and cost involved they have had less attention from researchers and commercial industry in general and specifically in terms of re-engineering.

2.6 Conclusion

This chapter has looked at the literature describing 4GLs (section 2.1). It covered the methods (section 2.2) as well as the key supporting concepts and techniques that others have used for extracting the underlying design and other information from them as needed (section 2.3). It also highlighted the difficulties that others have faced in finding appropriate metrics to measure ‘success’. In particular the challenges of reverse engineering 4GLs and some of the techniques proposed by other researchers were explored in section 2.4.

Finally, this chapter identified the research gap in syntactical approaches that targeted proprietary 4GLs for translation into 3GLs (section 2.5).

Chapter 3 Methodology shows how some of the main tools and techniques outlined above have been applied to start the process of Reverse Engineering the syntax of a specific 4GL (Uniface), using information derived from a specific real-life implementation (University of Essex - ESIS).

Chapter 3

Methodology

This chapter discusses the underlying methodologies and techniques used throughout the research.

It also covers the initial steps used to investigate the XML-like code extracted from the Uniface system, as extracted from the University of Essex ESIS system.

The methods used include deriving both a data model from the Uniface XML-like code (section 3.1) and also creating an abstract view of the the system's behaviour - what the Uniface system did, and therefore what the end system was supposed to do (section 3.2).

Section 3.3 discusses Component-Based Software Engineering (CBSE) and Component-Based Design (CBD) which were used as the underlying design processes across the full Reverse Engineering process.

Section 3.4 covers Compiler Framework Design since the Uniface grammar will need to be decomposed and then reconstructed as part of the Reverse Engineering process.

Finally, section 3.5 covers alternatives methods of ensuring the output of the reverse engineering process can be matched with the original for syntactical correctness and identifies the key reasons for the chosen method (chunking trees) as used in this thesis.

3.1 Deriving a Data Model from Extracted Uniface Code

3.1.1 Uniface's Built-in Database

Uniface uses its own built-in database to hold:

- developer’s code
- items needed to draw the user interface such as built-in components (for example text boxes, buttons and labels),

- the coordinates of elements on screen,
- the tab order (i.e. the information pertaining to pressing the keyboard 'tab' key), and
- obfuscated meta-data.

The ER model of the Uniface built-in database is not publicly available, as Uniface is a proprietary rather than an open-source language. On examination, it was found that some rows in the tables held incomplete code listings and it was also noticed that the segments that complete the row could be located in different tables which were named after the original table but with 'overflow' prefix. Therefore, it was impracticable to aim to directly extract the data model and structure of the database but modelling the exported XML-like code-base was felt to be a more plausible approach.

The XML-like file is a meta-data file that is exported using a Uniface export feature in the tools menu. This file contained artefacts about the system under development. This included sections that contained the implemented source code for the system being built with Uniface and relevant sections to identify code triggers. Code triggers are events that corresponds to user actions such as but not limited to mouse click, keyboard key press, content editing, components initialisation and screen resizing.

3.1.2 Using Unified Modeling Language (UML)

In the field of software engineering, UML is classified as a general purpose modelling language, which has been evolving over the last thirty years. However it has still not evolved into a formal specification language.

3.1.2.1 UML History

UML history 1990-2008 as taken from Zockoll et al. [7], has been shown in figure 3.1. The Object Management Group (OMG) adopted and standardised UML in 1997. UML is recognised by the International Organization for Standardization (ISO) as a standard [136]. UML was derived from previous object-oriented models prior to 1990 which were Booch notations (Booch, [137], Losavio et al. [138], Umeda [139] and Fayad et al. [140]) which discussed Object-Oriented Software Engineering (OOSE). UML then evolved from these in 1997 to form the most basic form of the language. The Rational Software Company adopted UML and created the first unified set of standards which was then proposed to the Object Management Group (OMG). A few minor revisions were then created to enhance UML. Nevertheless, UML was criticised for inconsistency and ambiguity by Reggio and Wieringa [141], Sibertin-Blanc et al. [142], Shah and Jinwala [143].

In 2005, UML received a major update (to UML2.0) to include new features and incorporated lessons learned between 1997 and 2005 in the software engineering paradigm.

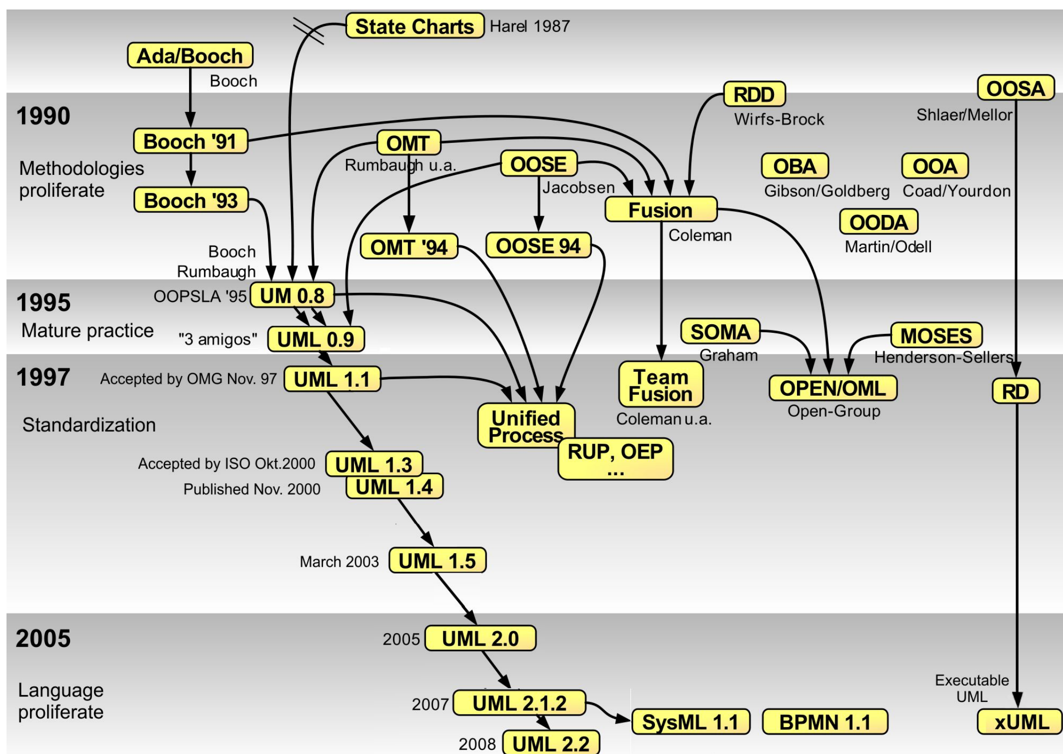

Figure 3.1: UML History 1990-2008 (Zockoll et al. [7])

The UML version as of December 2007 was UML2.5.1. [144]. References in this thesis to 'UML' are to this version.

3.1.2.2 UML Architecture and Components

UML is defined using a Meta-Object Facility (MOF) architecture to describe UML itself. This can be extendable using UML stereotyping.

Four main sections within the meta-model define the following aspects:

- notation,
- semantics,
- rules for the model, and
- layout.

UML defines two main types of diagrams, static and behavioural.

Static modelling techniques comprise:

- activity diagrams,

- state machine diagrams, and
- use case diagrams.

Behavioural modelling techniques covered by UML are:

- class diagrams,
- component diagrams,
- object diagrams,
- package diagrams, and
- deployment diagrams.

3.1.2.3 Difficulties of Using UML

UML has its own pitfalls when it comes to large systems according to France et al. [145]. Manual design review techniques can be challenging for large systems. UML supports Model-Driven Development (MDD) to implement and support the process of development at a minimum cost. However, understanding UML meta-modelling can be a complex task. Sometimes, therefore, developers do not have the full understanding of all aspects of UML to be able to run a transformation activity.

UML is not a formal language, which means its interpretation is subjective and therefore can be potentially ambiguous. In their 2014 study, about UML Class Diagram Formalisation, Alzahrani, Yafi et al. [12] found a risk of information loss when UML put in use to prototype and implement a system. Thereby, the newly built system would not perfectly match the legacy system in terms of implementation. In addition, using UML demands meeting substantial pre-existing requirements with a fixed scope for the implementation phase.

During the course of researching Uniface experimentally, it was found that Uniface stores the data and meta-data in a relational database. The author of this research performed a number of Structured Query Language (SQL) queries to attempt to extract artefacts from the database but this approach did not prove successful.

Nevertheless, in the absence of a valid ER model, a layer of abstraction is needed to model the Uniface syntax which is deeply hidden within the data store. Further detail about creating an abstract view of a Uniface system are provided in section 3.2 Creating an Abstract View of the Uniface System's Behaviour. Therefore, it was fundamental to use a formal representation such as relational algebra which is considered by a number of researchers to translate, and optimise SQL queries and perform fuzzy search on the data in the database (Ceri and Gottlob [146], Cyganiak [147], Cadoli and Mancini [148], Kim [149], Cao and Badia [150] and Galindo et al. [151]).

Extracting the ER model was not possible for the reasons set out above. For those reasons and because UML as standard can only provide an informal representation of the ER model, this

research has redirected its efforts towards using formal methods.

3.1.3 Modelling Approaches for this Research

UML is an informal approach to modelling the interior design of a system since there are many different valid ways a UML diagram can be interpreted or implemented in code.

Formal i.e. mathematical approaches define designs so that only a single valid interpretation is possible.

The research approach adopted was to use formal methods to enable formal verification where possible (Cabot et al.) [152].

The lack of formality inherent in UML led to the investigation of other formal methods such as relational algebra and grammar rules in the form of Extended Backus–Naur Form (EBNF).

3.1.4 Using Relational Algebra

Relational algebra, according to Cyganiak [147], is a mathematical language which supports set operations such as:

- selection,
- projection,
- set manipulations,
- union and difference,
- Cartesian product,
- include,
- joins, (including inner join, outer join, left outer join and right outer join),
- aggregation, and
- transitive closure.

The relational data model adopts the concepts of:

- table,
- tuple,
- relation instance,
- relation schema,
- relation key, and
- attribute domain.

In addition it promotes these types of constraints:

- key constraints,
- domain constraints, and
- referential integrity constraints.

This makes relational algebra appropriate for relational model databases.

Query languages which are built with relational algebra have a procedural programming style in common. The inputs and outputs for these languages are instances of relations. Unary and binary operators are the most used operators with relational query languages such as: Select σ, Project π, Union U, Set different $-$, Cartesian product $\times$, and Rename ρ.

These operations enable the translation, extendability, optimisation, interoperability between SQL syntax and relational algebra to exist. Translating SQL into relational algebra can reduce the overhead costs of nested 'join' operations and efficiently optimise the queries, according to Gingras and Lakshmanan [153] and Chatziantoniou and Ross [154].

Successful translating SQL into relational algebra, requires as a necessary condition, valid and cohesive data models both for the databases which Uniface either uses to store its meta-data and also for the data stores for the resultant application that are built with the Uniface 4GL.

3.2 Creating an Abstract View of the Uniface System's Behaviour

This section discusses the differences between traditional relational databases (e.g. SQL) and non-relational databases (e.g. NoSQL).

It discusses the uses and limitations of Non-relational databases, particularly in regard to searching for, storing and manipulating documents.

Relational database operations share the following common traits of being:

- Atomic,
- Consistent,
- Isolated, and
- Durable.

Each Create, Read, Update and Delete (CRUD) operation must complete fully and successfully or be rolled back to restore the database state to its state before the transaction was attempted. Also, each transaction runs in isolation of the other ones and (if complete) can be recovered in the event of failure.

By contrast, NoSQL (non-relational) databases are designed to abandon the concept of relational schema.

No-SQL databases share the traits of being:

- Basically available,
- Soft state, and
- Eventual consistency.

These properties mean that NoSQL databases are highly available databases even in an event of failure. The database maintains its consistency after applications input data, by replicating the data to nodes where replicas live. The soft state enables the data store to update without interactions from the external applications and will retain consistency as soon as the update is complete.

Whilst relational databases use tabular storage, NoSQL is designed to host different types of data using appropriate techniques which could vary from one database to another. Such data types are: key-value pair, graph database, document-oriented or column-oriented structures. Key-value pair databases use a hash table to store an array of keys and a pointer for each key to interlink to its value. The document-oriented database stores a document in one of these formats: XML, JSON or Binary JSON (BSON). Each document can have a different schema or no schema and a different set of attributes.

NoSQL is generally used mostly in big data and data analytic programs. Therefore, these databases are designed for scalability and growth of up to terabytes of data, accepting the fact that it can compromise the referential integrity that relational databases make use of.

The research in this thesis is aimed to transform the SQL structure of Uniface into a NoSQL structure in order to perform faster queries and allow a novel and new model for storing Uniface meta-data.

The original data structure of Uniface was not available to the researcher at the time this research was conducted. Therefore this research aims to make use of NoSQL to enable schema changes as it progresses. The data uniformity is not verified and therefore NoSQL has more flexibility than a regular SQL databases.

The author's research (based on Nayak et al. [155] findings), selected a document-oriented database model since it offers storage for documents which can have unlimited number of nested values in the same data format and store documents by avoiding splitting the document into a value-name pair. This type of database is available across platforms and is supported by a variety of operating systems. It is used when consistency and partition tolerance are a higher priority over availability, Nayak et al. [155]. Partition tolerance means that the system will continue to work notwithstanding network interruption or a failed node. Therefore, a diverse set of documents can

be stored in parallel as part of the same collection, thus aiding partition tolerance. These types of databases allow indexing to run on every property in the document regardless of key attributes.

Maintaining relationships between documents is viable either by referencing documents or embedding within them as sub-documents. Relationships of types 1:1, 1:N, N:1 or N:N are all feasible.

When a document is inserted into the database, an object identifier (object id) is created. The object id can either be defined by the developer or automatically generated by the database engine.

NoSQL databases are *schema-less* databases. Therefore, to maintain the database references, manual references or referenced relationships are used.

In Uniface, there are different types of custom procedures that correspond to events. Custom procedures are defined by users based on pre-defined function signatures that Uniface provides for each event type. Events are defined and triggered based on either user interaction (such as a click on the user interface element, typing in a text-box) or other internal triggers which are defined by the Uniface IDE.

The research for this thesis provides an optimisation technique which is used when dealing with NoSQL databases. The technique is called a *covered query*. To consider a query to be a covered query, it must only use fields which are part of an index. Additionally, all the fields which are returned by the query must be part of the same index. The database engine, therefore, will attempt to return the result without needing to evaluate the content of the document, which in return provides the relevant document properties. However, there are two conditions to run such a query. The first is that the yield field must not be an array, and the second is that indexed fields cannot be sub-documents.

There are other limitations when using NoSQL databases such as:

- Using indices can impact data changes operations such as Update, Insert, and Delete. Therefore, indices should be considered for collections when read requests are predominant,
- Indices are stored in RAM, so the indexed data should not exceed the size of available RAM, otherwise, the database engine might delete the index which will have a considerable negative impact on performance,
- Regular expressions, mathematical operations and *Where* clauses cannot be used with indexing.

The above index key limitations determine the number of documents that can be inserted into a specific collection.

The limitations as set out above were valid at the time of this research was conducted (2018). These might differ from one database vendor to another and could change over time.

To aggregate the results for a large volume of data, this research used MapReduce. This tool uses JavaScript functions to run a user-defined routine on the input data that is used as an argument when calling the function.

The next step was to investigate in detail, Uniface as a component-based development language. This is discussed in section 3.3 below.

3.3 Using Component-Based Methodologies Towards Reverse Engineering Uniface

3.3.1 Introduction

Component-Based Software Engineering (CBSE) is a practical methodology which is utilised to build fourth-generation programming languages and is used

- to improve overall cost estimation (Zia)[156], (Peeples)[157],
- for component validation (Dolado) [158], and
- for code generation.

All of the above reduce maintenance time, minimise errors, and improve code consistency (Brassard) [159]. Component-Based Design (CBD) and Component-Based Development Process (CBDP) are other terms essentially covering the same techniques.

This section describes the features, functions and usage of CBSE and in particular emphasises its benefits for code translation whereby a legacy system has been targeted in terms of components instead of by individual lines of code.

3.3.2 Component-Based Software Engineering (CBSE)

Component-Based Software Engineering (CBSE) is a branch of software engineering which promotes decoupling the software design elements of a system from its implementation. The CBSE approach enables shipping fragments of software in a compatible package.

Components are built and deployed to connect and integrate with the base system in a loosely coupled fashion. Components are compatible with software engineering design architectures such as: Event-Driven Architecture (EDA) and Service-Oriented architecture (SOA).

Services use components to build the base system and then convert each component to become a service in its own right, whereby the service characteristics become applicable to the component, according to Gfeller [160], Arad [161], and Ittel et al. [162].

Components enclose functions and package the set of functions in a module, service or package. Functions within a component are semantically related. A subset of these functions are publicly

accessible whereas other functions are implemented for internal use only (i.e. functions that call sub-functions to complete a task). Interfaces are designed to establish communication channels between components (Councill and Heineman) [163].

Each component is given a signature comprising its component name, input attributes and parameters.

Components are designed to become replaceable or substitutable if needed at compile-time or run-time. Even though a component might get replaced with another one, the signature of the component may remain as it is, as long as its name, and inputs remain the same. This signature is dictated by the interface that exposes the component (Martin) [164].

Components must also be able to be reusable if needed (i.e. the code base is shared across the system).

Uniface uses modelling, construction, and assembly to deliver the concept of a component-based system. Figure 3.2 below is an overview of the Uniface system from that perspective, as provided by Uniface [8].

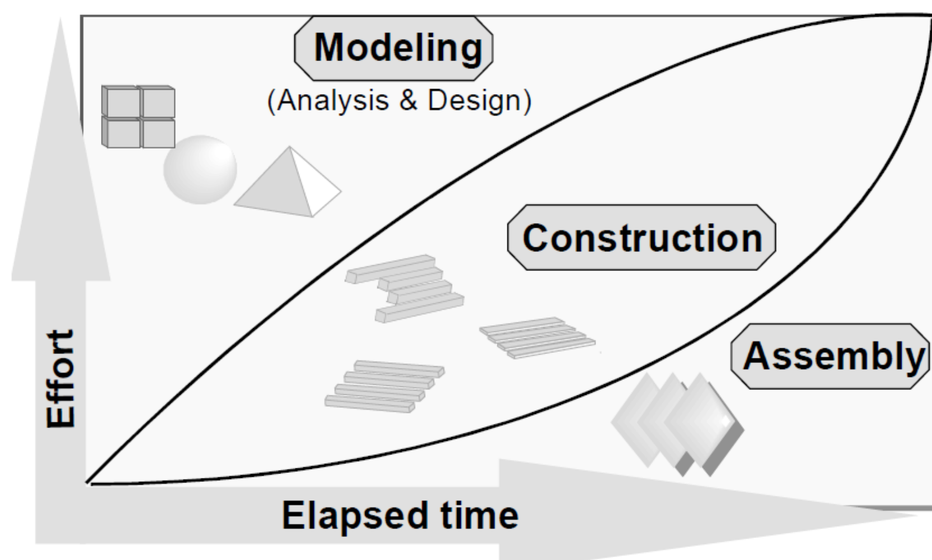

Figure 3.2: Uniface Activities (Uniface, 2000) [8].

According to Uniface [8], Component-Based Development Methodology is their preferred development methodology, whereas UML is seen as the perfect tool to model the language and its platform including the integrated development environment.

The most basic element in a component-based system is an object. Objects are real-world objects which are depicted using a programming language to describe the physical attributes of the object and to add semantics to its behaviour using routines and functions. Attributes and functions are held within the object using an object-oriented encapsulation technique.

Uniface implements the concept of data hiding in which a routine's logic and data-structure are kept hidden from the end user and only signatures are visible. This is an advantage of using component-based development as the understanding of storage and resource manipulation is not needed in detail. Instead, the communication between components is established without being a concern to the component developer.

From an architectural perspective, to provide a component-based system with effective data hiding mechanisms, it must hold each component in its relevant layer. An n-tier data architect, for instance, may use a three layer structure to represent a system. Those layers are:

- data layer,
- service layer, and
- presentation layer.

3.3.3 Advantages of Component-Based Design (CBD)

The following advantages of using CBD also relate to the process of modernising legacy systems which are built with 4GLs. In this research, the proof of concept implementation in section 8.4.3 adopts CBD techniques backed by the set of tools which are detailed in chapter 8.

The following were considered by Brassard in his US Patent [159] to be the most useful benefits of using and implementing a component-based system:

- Extending the system functionality is doable by adding new components to the system,
- Fixing a single component does not require any changes to any the other component,
- Components are substitutable, i.e. replacing a component does not require making any further changes to the system,
- Packaging the system is more fluent using a component-based system,
- Sharing a component's code base has become feasible,
- System layers are decoupled and maintained in isolation, and
- Scalability has been improved over similar types of systems.

CBSE is not a standard itself, but draws on UML standards. CBSE is constructed to optimise the design of system components. Component-based design outperforms entity relationship (ER) and data flow designs, which are designed for sequential processing. Component-based systems are designed specifically to benefit from these advantages, according to Narasimhan et al. [165].

Figure 3.3 below, from Uniface [8], shows the Component-Based Development Process (CBDP).

Component-Based Design - An iterative Process

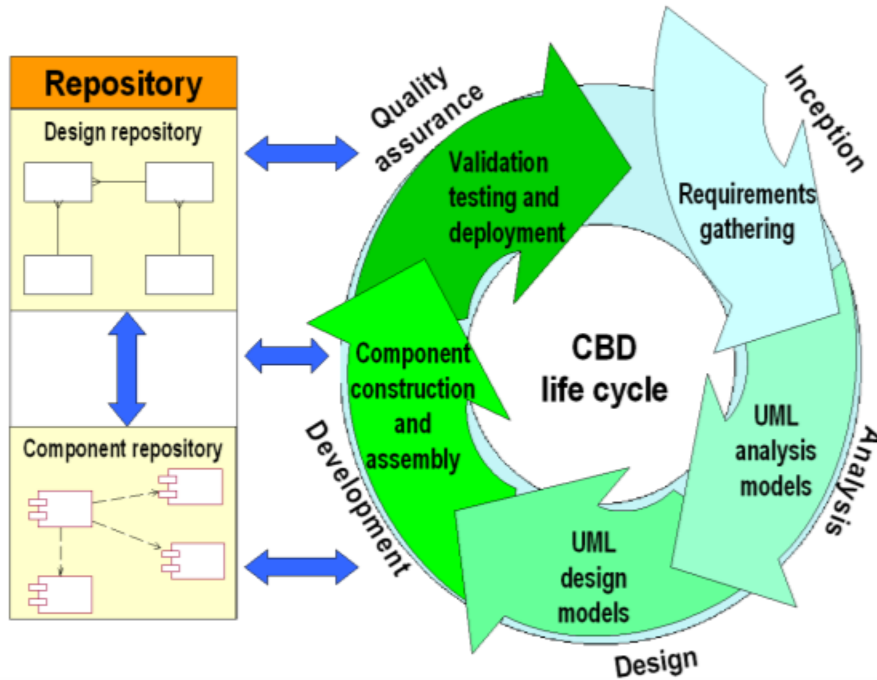

Figure 3.3: Component-Based Development Process (CBDP). (Uniface) [8]

The process is iterative, starting with business requirements gathering together with analysing and reflection on the findings. This is followed by UML analysis and design also using UML, which depicts the application attributes of the model. Following the Design phase, implementation (i.e. component construction and assembly) follows, which produces the software that matches the design and applies the business rules. Finally, validation and testing shows how the implementation meets the design (Quality Assurance) and the system, if fit, is deployed for use by the business. As wit most iterative design models, the current iteration (N) becomes the input for the next iteration (N+1) of the particular system.

This research does not deviate from the concepts which were constructed to build Uniface itself. However, it aims to formalise the design phase and translate the syntax using compiler-compiler design methods. Therefore, CBD delivers incrementally, where each increment is an independent delivery which constitutes a package of a set of components.

3.3.4 Incremental Development - Advantages, Principles and Techniques

3.3.4.1 Incremental Development Advantages

The research set out in section 8.4.3 benefited from the advantages that came with this *agile* process. It maintained the development process throughout by delivering chunks of deployable pieces of code incrementally as well as allowing individual tasks to have been built in parallel. It also enabled the execution of test methods with less code than monolithic applications, which reduced the risk of making many changes together.

Incremental development enables different teams to work on the same project at the same time in parallel. It also reduces overhead cost but requires well formed plans and well prepared software development personnel. In addition, development standards must be declared in advance and - where team-working is used - all teams must adhere to them. This research, although solo work, followed this practice.

3.3.4.2 Incremental Development Principles

Incremental development principles as set out by Hibbs et al. [166] who was referring to the CBSE as 'lean' software development, set out the minimum viable product (MVP) is delivered throughout the process, which consists of the following steps:

- Setting the objectives.
- Depicting the architecture blueprint.
- Drawing up a plan.
- Adhering to the defined standards.
- Reserving the needed resources and utilising the available resources.
- Introducing a set of selected functionality to develop.

Hibbs et al.s essentially sets out the same steps as in the CBSD model 3.3 Component-Based Development Process (CBDP). (Uniface) [8] above, albeit using slightly different terminology.

In addition to these steps, a priority list must be defined for the set of functions and components that needs to be delivered in each iteration. Defining a use case priority is derived from the business's/product owner's values and priorities. These needs to be quantified either using a points-based system or a suitable different measure which should be agreed by the product owner.

To mitigate the risk of exceeding the project budget, Barki et al. [167] recommended to start developing the inexpensive components. Doing this will improve the calibration process of determining the cost and time for the future components.

In this research, the dependencies were specified in advance to build the base components first and influence the order of the next package for development.

According to Barki et al. [167] measuring system complexity is a recommended step, prior to starting to build the base components. These measures should be applied to the physical resources and logical resources such as hardware, networking, databases, and the available source code. The less complex components have less probability of failure. Therefore, it is a valid justification to start the first iteration with the less complex packages or sub-systems.

At the end of each iteration, the delivery-ready packages must be tested with unit testing and regression testing. Regression testing involves repeating all the tests run so far, to ensure that the new component or unit added has not introduced or uncovered any bugs elsewhere in the system as a whole. It can be complex and time consuming and is best automated if possible.

In addition, they can be checked against the pre-defined set of quality assurance measures. In addition to quality assurance, end-users feedback (or business user representative if end users are inaccessible) should be also taken into account for the next iteration's requirements. Then, new analysis and design tasks should be established to address new changes.

3.3.4.3 Incremental Development Techniques

Two main techniques are discussed here - prototyping and testing.

Prototyping

Prototyping is an additional technique that takes place during the phase of design and analysis. The most common techniques are exploratory and experimental prototyping.

- Exploratory Prototyping

Exploratory prototyping involves creating mock-up screens for the user to use and gain experience in using the graphical user interface to run limited functionality which simulates a business use case; this allows users to provide feedback to the development team.

Exploratory prototyping works with Joint Application Development (JAD) according to Liou and Chen [168] and Duggan Thachenkary [169].

- Experimental Prototyping

In experimental prototyping, the researcher/developer creates a proof of concept (POC) product which affirms that the potential for implementing this product is achievable. Therefore, the proof of concept is created using different component designs, technologies or architecture designs.

Moreover, the experimental techniques are not only subject to a variety of implementations but also to packaging and deploying the source code.

Duggan and Thachenkary's paper [169] sets out good practices for experimental prototyping:

- Selecting the components that need building during the POC phase,
- Scoping the prototype by the set of business requirements which relate to the selected components,
- Implementing the core functionality for the selected components,
- Running the solution and providing documentation for the outcome,
- Validating, verifying and comparing the results, and
- Identifying the most appropriate implementation to use.

Experimental prototyping is also a useful tool to assist the decision makers. Experimental prototypes can be suitable to be 'built onto' for the real production systems as opposed to 'throw-away' prototypes which are used to 'buy knowledge' but are not fit or suitable for evolutionary further development.

In the research for this thesis, the experimental prototyping approach was selected.

Testing

Testing the components is the next phase. Three main classes of testing should be carried out, in the order as listed:

- White-box testing,
- Black-box testing, and
- Non-functional testing.

They are used to test the functionality of a component-based system, and discussed below.

- White-box Testing

According to Khan, M. and Khan, F., [170]:

> "White Box Testing Technique: It is the detailed investigation of internal logic and structure of the code. In white box testing it is necessary for a tester to have full knowledge of source code."

Because of this detailed code-level knowledge requirement, white-box testing is only suitable for relatively small components or units, and not large systems or sub-systems.

White-box testing requires deep understanding of the component. It requires a good understanding of the business requirements and the internal implementation that reflects the requirements. In white-box testing, the structure and behaviour of the component is targeted.

White-box testing tests the related components individually and then a set of regression tests are run, which test how the components interact with each other. Ideally, testers with a good knowledge of the components do this type of testing rather than the implementer(s), to avoid

the inherent bias that the developers have when testing their own developed code. A developer's mindset is “I want my creation to work” whereas a tester has the mindset “I want to try and break it.”

- Black-box Testing

Khan, M. and Khan, F., [170] defined black-box testing as:

> “Black Box Testing Technique: It is a technique of testing without having any knowledge of the internal working of the application. It only examines the fundamental aspects of the system and has no or little relevance with the internal logical structure of the system.”

In black-box testing, just the interaction between the user and the components are targeted for testing. Therefore, the component interface and interactions are the test subject, treating what goes on inside the component as an unopened ’black box’.

Firstly, the system with the deployed components should run and show compliance with the requirements. Then, additional components are added and the same tests will run again to ensure that the functionality has not been affected.

- Non-Functional Testing

This is the final stage of testing, after black-box testing.

In non-functional testing, performance tests, stress tests and concurrency tests (Gunda and Devta-Prasanna [171]) are run and the results are validated against the original requirements.

- Automated Testing

Automated testing can be used to save time and cost. However, regression testing must be run first to ensure that the system works with the new components in all possible scenarios and to ensure performance and behaviour has not degraded (i.e. regressed), then the behaviour can be automated for repetition.

3.3.5 Adapting Existing Tools or Building Bespoke Tools?

This research used both compiler-compiler tools and also compiler-generator tools and libraries. These tools were developed to simplify the process of creating a compiler.

Assuming ‘off-the-shelf’ tools (e.g. a parser generator for Uniface) are not known or available, one approach is adapting a tool that has been created for a different specific language, but which is not yet available targeting Uniface. A second possible approach would be to write a parser generator tool from the ground up.

The first approach would be the preferred one if possible, because writing a compiler-generator would be a protracted task. However, Uniface has not been shipped with a parser that could

analyse the source code of applications written with Uniface (such as ESIS). Neither was a parser available for transforming or translating Uniface language's syntax. Therefore, the author of this research built novel grammar rules to compile a sub-set of Uniface syntax using both ANTLR (section 8.4.1) and Gold Parser (section 8.3.2).

3.4 Compiler Framework Design

The process of designing a compiler framework - in this case in the context of reverse engineering Uniface's grammar - includes the steps needed to achieve the final product such as generating the parser and the Abstract Syntax Tree (AST) as well.

The authors in the following research publications, Fourment and Gillings [172], Malloy et al. [173], Derezińska and Szustek [174], Singh et al. [175], Hanson and Proebsting [176], Parr and Fisher [130], Lilis and Savidis [177] and Pawade et al. [178] surveyed some of the tools that were used mainly with the C# language. This was the implementation language for the main parts of code resulting from this research and guided this research in terms of the parser-generator tools used such as ANTLR.

3.4.1 Introduction to Compilers, Parsers and Lexers

Typically, all compilers process input using six phases:

- lexical analyser and generator,
- syntax analysis,
- semantic analysis,
- generation of intermediate code which produces optimisation ready code,
- optimisation phase, and
- binary code generation,

as described by Aho et al. [179].

Grammar rules can nest within other rules. The rules guide the compiler through the input data and enable validation without backtracking through the input.

Rules are written using Regular Expression Parser techniques which feed the Lexer. The Lexer scans the input and finds matches to output as lexemes (also known as tokens, as described in section 3.4.5). Input is read character by character and then regular expression rules are applied to the scanned text. When the parsed file has been generated, it creates a tree of lexemes.

3.4.2 Parsers

Parsers generally comprise a lexer and scanner or tokeniser as shown in figure 3.4, although some parser-generators are *scannerless* i.e. do not use the first step, lexing, according to Salomon and Cormack [180].

Figure 3.4 (Parser Structure: Lexer, Tokeniser, and Parse Tree (Author's Diagram)) below shows the key functional elements of a parser.

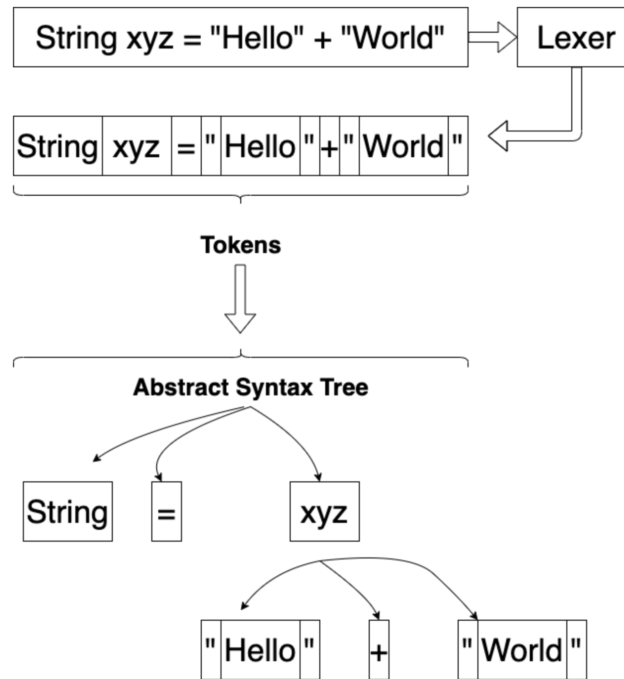

Figure 3.4: Parser Structure: Lexer, Tokeniser, and Parse Tree (Author's Diagram)

Some toolsets offer a lexer generator independently whereas the majority offer both, the lexer and parser generator.

3.4.3 Abstract Syntax Trees (AST)

Compiler-compiler generators use the rules as an input to generate the parse tree. Then the parse tree is used with the input code to extract the AST. Therefore, the AST contains only the tokens and grammar rules which are used in the program.

This research investigates the Uniface AST, as reversing Uniface requires extracting rules from existing code. Therefore, knowing the full set of rules which were created at the time when Uniface was designed is not feasible.

Figure 3.5 shows an example of the AST which is based on the following pseudo-code snippet:

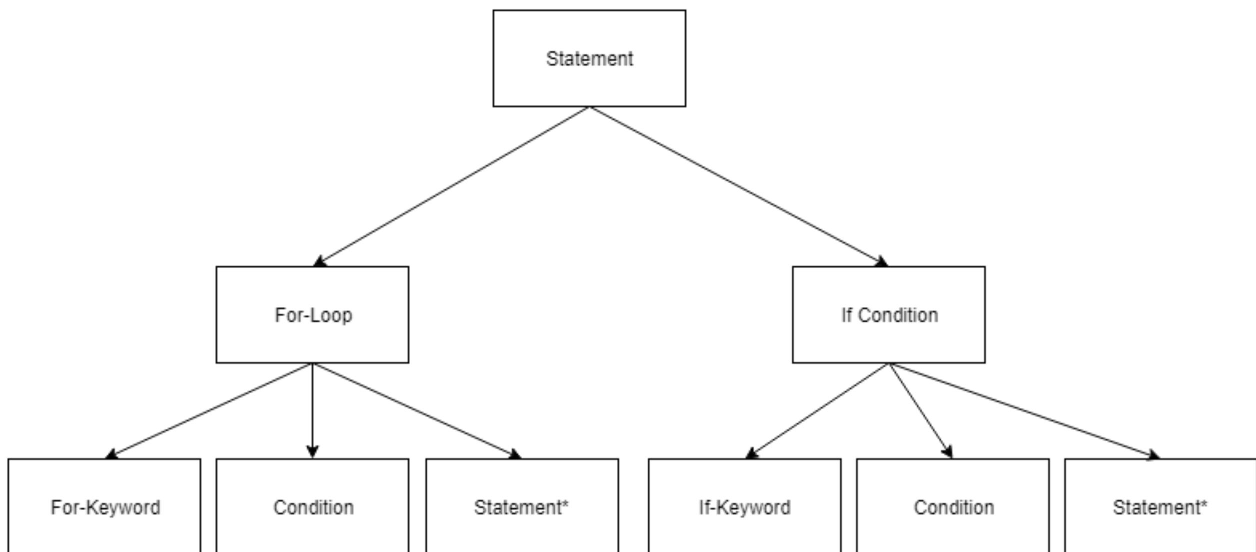

Figure 3.5: Parse Tree, Abstract Syntax Tree

```
1 For [start index .. end index]
2 [Statement] ..
3 If [Boolean Condition]
4 [Statement] ..
```

The line numbers have been included in the above snippet purely for ease of explanation and did not form part of the code language.

Line 1 contains the keyword 'For' followed by a starting index and ending index for how many iterations the code within the block will run. Line 2 contains a statement or a list of statements which then will show on the AST subject to the grammar rules.

Line 3 represents an 'If' statement which contains the keyword 'If' followed by a Boolean condition and then line 4 contains a statement, or a number of statements, that will be executed if the condition is true.

The AST is a syntactical representation of the order of execution of the snippet where deconstruction in AST form can replace the input code by providing a tree structure of it.

3.4.4 Parser Techniques

LL (Left-to-Right, Left-Most derivation)

- LL parser constructs the parsing tree using top-down approach where the scan starts with the starting symbol then the tree is traversed to expand non-terminals.

LR Parser (Left-to-right Right-most derivation)

- LR parser constructs the parsing tree using a bottom-up approach. Where the scan ends with the starting symbol then the tree is traversed to compress terminals.

LALR Parser (Look-Ahead LR parser)

- LALR uses LR parser with look-ahead to optimise the parsing tree by merging similar rules into one.
- A parser's name may be followed by a numeric (k) where k denotes the number of look-ahead of tokens.

Other points to note are:-

- Grammars can have semantic rules in addition to attributes. The parser checks for LL(1) conflicts and completeness.
- LALR(1) are less able to resolve errors and provide error recovery solutions.

3.4.5 Grammars

Grammar is a set of rules that instructs the interpreter during the process of interpreting the developed code and constructing the parse tree that represents the grammar.

According to Lämmel and Verhoef [24], grammar is defined as: "Grammar is the formal specification of the syntactic structure of a language."

Grammars are omnipresent in the meta-programming field. A grammar can be used:

- to define the syntax of a programming language,
- to define a common pattern used as an input string,
- to define the structure of standard documents such as Extensible Markup Language (XML) or a Document Type Definition (DTD).

Backus-Naur Form (BNF) and Extended Backus-Naur Form (EBNF) are interchangeably used to describe a particular language grammar. BNF and EBNF use the Chomsky definition [181] of formal grammar where tokens are classified as *terminals* and *non-terminals* and then production rules determine how to interpret a rule, according to Cook and Wang [182].

There are two types of grammar. Regular grammar Yu [183] and context-free grammar Charniak et al. [184], Aho and Ullman. [185] and Cremers and Ginsberg [186]. Regular Grammar describes a language using regular expressions. But when regular expressions are not enough to parse the language, a context-free grammar can work, using recursive features. HTML, for instance, is a context-free language, so the language offers optional parameters that can be customised and extended. These custom attributes can prevent the parser from correctly scanning text containing more than just regular expressions.

Both grammar styles can be used with parser-generator tools. Gardens Point LEX (GPLEX) as described by Gough [187], for instance, uses the LALR(1) parser and it generates bottom-up parsers. It will operate with context-free language. GPLEX generates a lexical scanner in the C# language and improves the typical LEX software tool with the ability to utilise full 21-bit Unicode scanners. GPLEX mainly works with the Gardens Point parser generator but it also can work with COCO/R and other custom parsers.

3.4.6 Parser Generator Tools

The research in this thesis has extended the ability of using such tools to cover the characteristics of Uniface.

3.4.6.1 ANTLR

Another Tool for Language Recognition (ANTLR) (Parr - ANTLR4 Reference, 2013) [188] is a Java tool that enables generation of parsers using ALL(*), Parr et al. 2014 [189] in Java, and C# as well. ANTLR supports other languages which are not of interest to this research. ANTLR can be extended using *Targets* that provide a mechanism for it to generate the parser in a specific language. It is used for transforming object-oriented languages such as C# and Java as shown in Bruneliere et al. [190], Wulf et al. [191], Eysholdt and Behrens [192], Parr, 2008 [193].

Using any parser generator such as ANTLR requires skills in development and a good understanding of the target system's language. Some languages may be covered and supported by official ANTLR releases, whereas others may not.

ANTLR - Grammars written for ANTLR v4 [194] offers a list of grammars for different languages. Some of the grammar files may not contain an up to date representation of the language. However, it might fully contain all the grammar rules of an older version of the language. Uniface, for example, had not been evaluated or made available yet at the time this research was conducted.

ANTLR generates *Listeners* and *Visitors*. The former allows the developer to travel the syntax tree nodes (either abstract or before abstraction), whereas the latter contains triggers that fire up the rule custom code when a grammar rule is matched with the parsed segment of input.

ANTLR rules are logically divided into two categories, lexer rules and parser rules. Each set of rules can be saved into a separate file.

3.4.6.2 Coco/R

Coco/R is another compiler generator tool. It uses Deterministic Finite Automaton (DFA) and EBNF grammar to generate a scanner and parser which supports Unicode UTF-8 characters according to its manual, as authored by Mössenböck [195]. Coco/R uses an attribute grammar that can either be stored as a file, or provided as an input stream into Coco/R. Coco/R allows the developer to use a custom scanner instead of the one provided by the tool.

3.4.6.3 Gold Parser

Gold Parser, Cook [196], also supports the C# language in addition to many others such as D, Java, Pascal, Python, Visual Basic.NET and Visual C++. Gold Parser is a Look-Ahead Left Right (LALR) parser which supports an advanced error recovery technique and an intuitive reporting user interface. Importantly for this thesis's research project, Gold Parser also supports Unicode characters.

This research also benefited from the features that Gold Parser offers to write grammar, test the grammar and generate a skeleton program in C# which can be extended to deliver the goal of providing a mechanism to assist in translating the syntax of Uniface and other 4GLs.

3.4.6.4 Grammatica

Tools like *Grammatica* for instance Cederberg [197], use LL(k) grammar to overcome this hurdle. Grammatica generates the parser at run-time. This technique enables testing of the grammar without the need for custom code to be written for that task.

Grammatica uses logical separation to separate the header, tokens and production rules. All three sections are written in the same grammar file.

3.4.6.5 Hime Parser Generator

Using the LR parsing technique *Hime Parser Generator* generates a parser with the support of template syntactic rules and context-sensitive lexing. Hime uses a generalised LR algorithm as set out by Economopoulos in his PhD Thesis [198], to parse any context free grammar.

Similarly to ANTLR, Hime offers a library of pre-defined grammars for languages. Uniface does not exist in its list [199].

Hime also supports context sensitive lexing. For example, in C# the keyword *get* could mean yield a value of a property but it could also be a variable name depending on the context, as the code snippet below shows.

```
1       public MyProperty { get { return someField; } }
2       public void DoStuff() { int get = 1; }
```

Solving such a problem in ANTLR and Hime is similar to specifying the rules for the exact token match first and then specifying the rules for all other tokens which are constructed of symbols such as:

```
1       SYMBOL -> [a-zA-Z_] [a-zA-Z_0-9]* ;
2       context accessors { GET -> 'get' ; }
```

3.4.6.6 Loyc LL(k) Parser Generator (LLLPG)

Loyc LL(k) Parser Generator (LLLPG) uses the LL(k) parsing technique. The motive behind LLLPG is to overcome the poor performance of ANTLR Targets for C# as Bovet and Parr claim [200].

3.4.6.7 Ironmeta, OMeta, Pegasus

Some other tools use Parsing Expression Grammars (PEG), such as *IronMeta* - Laurent and Mens [201] which is built on top of *OMeta* Warth and Piumarta [128] and *Pegasus* Knöll and Mezini [202].

3.4.6.8 Sprache and SuperPower

Some other platforms and libraries, like *Sprache* [203] for example, and *SuperPower* which is built on top of Sparche [204], are made lightweight to deliver parsing capabilities directly in the targeted language. Other examples, available via GitHub. are: *ParseQ* [205], *Parsley* [206], and *Pidgin* [207].

Some other tools use Java as the main language for the generator such as:

- *AnnoFlex* - Czaska [208],
- *ABNF Parser Generator* (APG) - Thomas [209],
- *BYACC/J* - Corbett [210],
- *CookCC* - Yuan [211],
- *CUP* - Hudson et al. [212],
- *Jacc* - Jones [213],
- *JavaCC* - Copeland [214],
- *JFlex* - Klein et al. [215],
- *ModelCC* - Quesada et al. [216],
- *PetitParser* - Kurs et al. [217], and
- *SableCC* - Gagnon et al., 2002, [218], Gagnon and Hendren, 1998, [219].

3.5 Determining the Syntactical Correctness of Translated 4GLs

Comparing the syntax of programming languages has been an under-researched field. The lack of studies has been driven by the absence of existing and valid criteria for describing and measuring

syntactical features of a computer programming language. Each programming language conforms to its pre-designed grammar. In principle, language engineers could compare the two language's grammar rules and identify all syntactical forms of coding statements to highlight similarities or differences. The author of this research could not find any evidence to claim that this approach had been either considered or investigated.

Another approach would be for the input syntax could be compared to the expected output syntax where the expected output syntax is also specified and the two compared.

Researchers have investigated three potential methods to asses the correctness:

- Screen output,
- Unit testing,
- Chunking trees.

Each is considered briefly in turn.

Screen output: This is a method that tests the output on the user screen when given a specific input. To run the test, a developer or tester runs the same user interface forms in both legacy and translated languages. The tester inputs the same data into both user interfaces and records the output. The test is passed if the output on both screens matches. This method can also be automated using the appropriate tools, according to Harrison and Berglas [220, 75].

This approach tests both syntax and semantics together but not separately. It is language agnostic.

Unit testing: Unit testing is a verification method where tests are run on a segment of program, to verify that its implementation conforms to its design. A tester writes code to test the expected output based on static, carefully chosen representative and boundary input. The input is defined in unit tests and the output is verified by the testing framework, within which the unit tests have been written. This approach is used by Broløs et al. [221] where according to the authors unit tests were semantically used for "ensuring that each component behaves as intended".

This method is very expensive to implement in terms of needing skilled testers, their cost, and the time taken to develop, validate, and perform the tests, and was found by Ellims et al. [222], and by Runeson [223].

Unit testing has only been used to test system behaviour and the author of this thesis has found no evidence to support unit testing having been used for syntactical approaches.

Uniface does not support automated unit testing as far as the author can determine.

Chunking trees: This is a novel method, used in this research to compare ASTs of the input language syntax to the output language syntax. This method reduces the number of nodes that ASTs present and preserves only nodes which are relevant to the comparison.

Chunking trees can only be used to compare two programming language's syntax whereas the other two methods above are best for the comparison of semantic behaviour of code.

Chunking trees can be used with any type of programming language based on their AST.

Chunking trees have been adopted by the author of this research, as the chosen method to compare the syntax of the input programming language being reverse engineered to that of the output programming language being generated. This is because the chunking trees approach:

- can be used to compare the syntax of two computer languages,
- reduces the cost, effort, and complexity compared to writing unit tests,
- reduces the number of nodes of the source and target language ASTs. This simplifies the comparisons and keeps it relevant by reducing false positives, and
- is equally applicable to both source and also target language, independently of the source and target language's grammar rules as well as the language of the parser.

3.6 Conclusion

This chapter has described the methodologies used in building the fourth-generation programming languages and the integrated development environments IDEs which offer an interface to build, interpret, and validate the syntax for the developers who will use these languages to build applications with. It surveyed a list of tools which were used with domain specific languages and discussed the opportunities for using the observed techniques to reverse-engineer Uniface and the other languages that shared similar characteristics. It also covered the chosen method (chunking trees) used to determine syntactical correctness of the output.

The following chapters

- Chapter 4. Implementation I - Extraction, ER and Relational Algebra Modelling, and
- Chapter 5. Implementation II - Reverse Engineering Uniface Schema

aim to depict the problem statement and use the component-based system methodology to research the potential in extracting Uniface artefacts and discover the underlying Uniface language syntax and its grammar rules accordingly.

Chapter 4

Implementation I - Extraction, ER and Relational Algebra Modelling

This chapter covers the start of the implementation process, based on XML-like code extracted from Uniface and then cleaned of deleted data. An Entity Relationship (ER) model of data structures can be created (an example of this is provided) and that model can also be represented using relational algebra. These are initial steps towards the author's proof of concept system to take a 4GL (Uniface) and reverse engineer it into an object-oriented (e.g. UML compliant) code structure.

This chapter comprises:

- System Architecture (section: 4.1),
- Extracting Uniface XML-like Code and Static Model Extraction (section 4.2),
- Using Relational Algebra to Represent the ER Model (section 4.3),
- Parsing Uniface Data and Meta-Data (section 4.4),
- Encapsulated Document Object Model (EDOM) (section 4.5),
- Why Unicode is Needed for the Uniface Character Set (section 4.6).

Further aspects of the design of the system are covered in

- Chapter 5 Implementation II - Reverse Engineering Uniface Schema,
- Chapter 6 Implementation III - Extracting Uniface Elements,

- Chapter 7 Implementation IV - Grammar Formalism.

4.1 System Architecture

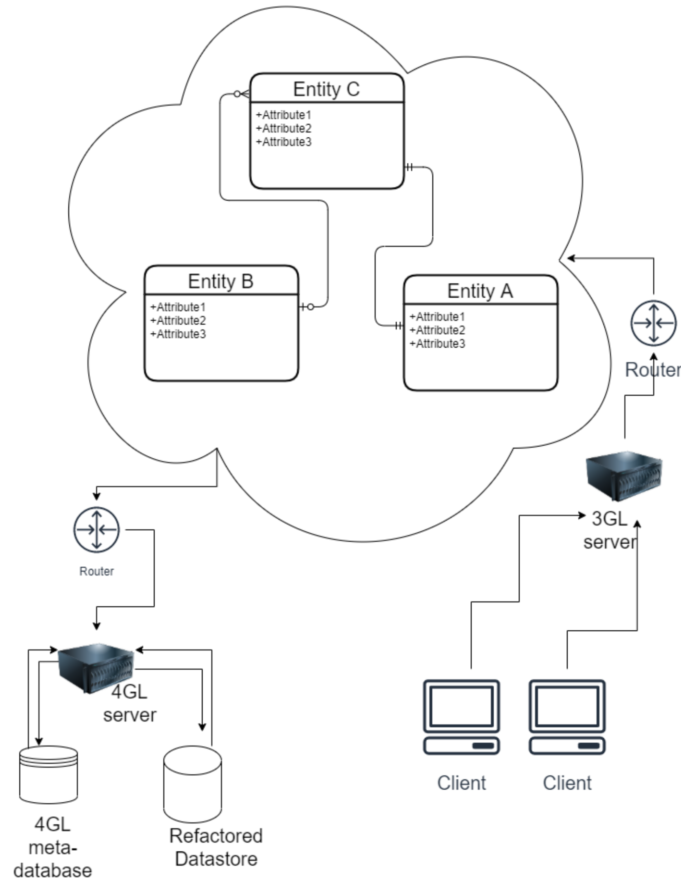

Figure 4.1: System Architecture - High Level Generic

Figure 4.1 (System Architecture - High Level Generic) above, shows a generic outline of a high level view of the proposed system architecture for the author's proof of concept system.

The following sections in this chapter, show how the Uniface XML-like code, as extracted from the ESIS system, had to be processed in order to obtain the underlying embedded Uniface code and data schema from within it.

4.2 Extracting Uniface XML-like Code and Static Model Extraction

Commercially sensitive date has been redacted.

4.2.1 Extracting Clean XML Data from Uniface

Uniface internally holds data as XML. This applies to the data that is either stored internally in its database subject to SQL queries for access or the data that is represented as nodes and is subject to export in XML files via the Uniface export utility.

A feature of Uniface is that it holds any deleted fields and keeps records of those in the database. Once the export function has been triggered, Uniface also includes references to deleted fields within the exported XML-like data. This adds noise which means that unwanted fields must be cleaned up (i.e. removed) before the exported file can become usable. Fortunately, Uniface offers command line arguments and an interface menu button to clean up the data and remove unused entities. Data exported following this process has been referred to as ‘cleaned data’.

4.2.2 Static Model Extraction

The example below sets out how a snippet of Uniface extracted XML code was deconstructed to reveal the coordinates of two forms, each holding a data structure (data entity) and the relationship between the data structures. The two forms/data structures are described as ‘inner’ and ‘outer’ forms in the text below.

The relations between the entities in the exported cleaned XML code of the Uniface model of the exported system are presented as a single XML file. Within this file is a <DAT> node that has a name attribute equal to “FORMPIC” representing the inner and outer entities relationships.

To understand which entity contains which, the geometric positions of the entities have to be measured. In the XML listing below, the width and height of ‘ASR_DET’ entity are defined within the XML with two variables ‘WID’ and ‘HEI’

The listing below is a transcribed version of a single continuous row of a snippet of exported XML (formatted for readability).

```
1 <DAT name="FORMPIC" xml:space='preserve'>
2 &uFRM;TYP=E&uSEP;NAM=PR_MM.[redacted]&uSEP;WID=86&uSEP;HEI=33&uSEP;HOC=86&uSEP;VOC=3&uFRM;
3 &uFRM;TYP=L&uSEP;NAM=PR_ID.PR_MM&uSEP;WID=5&uSEP;HEI=1&uFRM;
4 &uFRM;TYP=F&uSEP;NAM=PR_ID&uSEP;WID=10&uSEP;HEI=1&uFRM;
5 &uFRM;TYP=L&uSEP;NAM=C_NATIONALITY.PR_MM&uSEP;WID=13&uSEP;HEI=1&uFRM;
6 &uFRM;TYP=F&uSEP;NAM=C_NATIONALITY&uSEP;WID=4&uSEP;HEI=1&uFRM;
7 &uFRM;TYP=L&uSEP;NAM=SURNAME.PR_MM&uSEP;WID=7&uSEP;HEI=1&uFRM;&uFRM;TYP=F&uSEP;
```

```
8 NAM=SURNAME&uSEP;WID=30&uSEP;HEI=1&uFRM;
9 &uFRM;TYP=L&uSEP;NAM=OTHER_NAMES.PR_MM&uSEP;WID=6&uSEP;HEI=1&uFRM;
10 &uFRM;TYP=F&uSEP;NAM=OTHER_NAMES&uSEP;WID=40&uSEP;HEI=1&uFRM;</DAT>
11 <DAT name="PERF">#DEF</DAT>
```

The code snippet in the listing above, shows Uniface XML code that represents a single screen. The screen displayed student information from two entities ‘ASR_DET’ and ‘PR_MM’. The author of this research both created this example and also made changes in the position of the entities on screen to observe the variables change value in the exported file.

The outer entity has a horizontal coordinates WID of value set to $WID = 7$ and a vertical coordinate HEI set to $HEI = 1$. The inner entity width $WID = 10$ and $HEI = 1$. These entities overlap and this overlap means that Uniface will establish a relationship between the database tables which these entities represent when Create, Read Update, Delete (CRUD) data routines have been called. Uniface uses this information to look up the relationship between the entities ‘ASR_DET’ and ‘PR_MM’ respectively.

The possibilities for a relationship declaration in Uniface are:

- Defined in the model,
- Hard coded,
- Inherited from generic libraries, or
- Drawn on the screen as inner and outer entities.

The aim of the system as designed here, is to convert the model extracted from Uniface into an ER model. This model could then be used to generate the architecture of a 3G modern object-oriented system.

It was found that Uniface records the list of all the relationships between the tables it contains. This table is called “UCRelsh” and also links to the current Uniface meta-data table that is named “UREPOSITORYNAME95VERSION01”, where the number indicates the installed version of the Uniface application.

Figure 4.2 Uniface Relationship Builder Tool below illustrates the relationship using the Uniface user interface.

The following SQL statement was generated by Uniface to get the student's records based on the two tables that were created in Uniface (figure 4.2) and exported to a file named ‘majd_testẋml’.

```
1 select top 100 * from pr_mm p inner join asr_det a on p.pr_id = a.pr_id
  ↪ where p.other_names = 'majd'
```

This SQL statement was captured on the server instance of SQLServer by listening to the incoming requests using the native monitoring tools of SQLServer as set out in Jorgensen et al. [224].

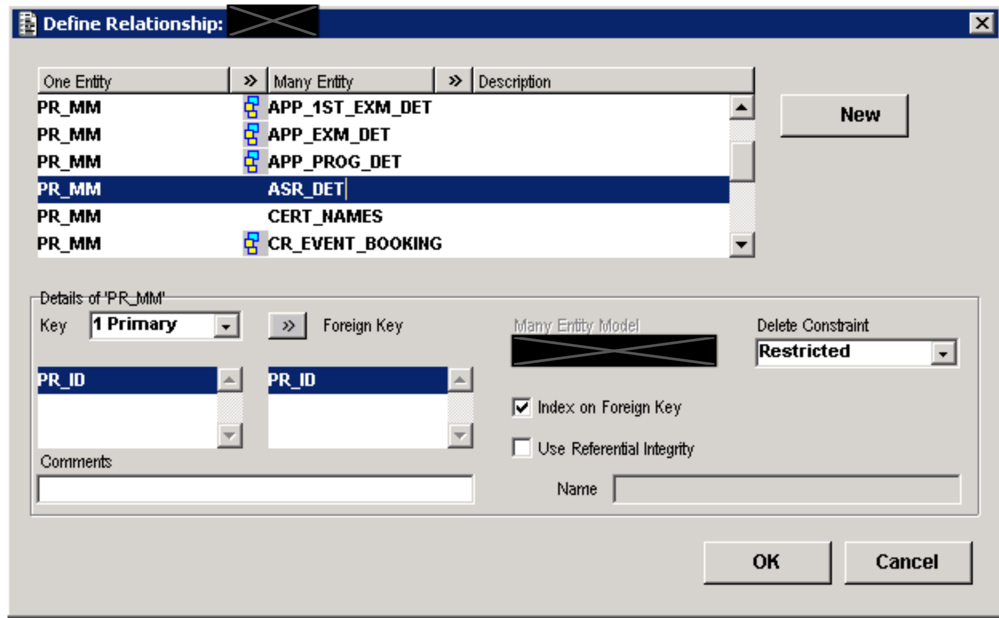

Figure 4.2: Uniface Relationship Builder Tool

Entities

ASR_DET	**REGULAR ENTITY**
ASR_KEY	ATTRIBUTE
PR_ID	[PK]ATTRIBUTE Type: NUMERIC
PR_MM	**REGULAR ENTITY**
PR_ID	[PK] ATTRIBUTE Type: NUMERIC
ASR_Key_1	[PK]

Table 4.1: Entities ASR_DET and PR_MM, their Keys and Attributes Types

The matching entity relationship for the data entities ASR_DET and PR_MM (as given in table 4.1).

The relationship between ASR_DET and PR_MM is provided by ASR_PR ASR_PR : REGULAR RELATIONSHIP ASR_DET ONE MANDATORY to PR_MM ONE MANDATORY

According to Chen [225] “The entity-relationship model can be used as a basis for unification of different views of data: the network model, the relational model, and the entity set model.” See figure 4.3 below for the ER diagram that describes the M:N relationship ‘ASR_PR’ between the two student entities ‘ASR_DET’ and ‘PR_MM’.

In addition to the relationship table that Uniface maintains, it holds the records for the primary and foreign keys in the “ucKey” table.

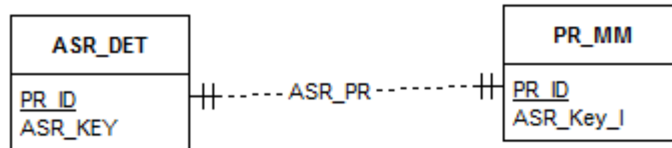

Figure 4.3: ER model for Student Data Entities ASR_DET and PR_MM

Figure 4.4 below shows the SQL query which extracted the primary keys for both PR_MM and ASR_DET entities:

```
SELECT TOP 1000 [utimestamp]
      ,[u_stat]
      ,[u_vlab]
      ,[u_tlab]
      ,[u_kseq]
      ,[u_ktyp]
      ,[uvers]
      ,[udescr]
      ,[u_doc]
  FROM [urepos9501].[dbo].[uckey]
  where u_vlab ='' and (u_tlab ='asr_det' or u_tlab ='pr_mm') and u_ktyp ='P'
  order by u_tlab
```

Figure 4.4: Student Data Entities ASR_DET and PR_MM - Primary Key SQL Query

The output from that query are shown below in figure 4.5:

	utimestamp	u_stat	u_vlab	u_tlab	u_kseq	u_ktyp	uvers	udescr	u_doc
1	2000-11-23 15:33:04.000	NULL		ASR_DET	1	P	NULL	NULL	<unimeta><U_FLABS>ASR_KEY</U_FLABS> <UVALKEY>T</...
2	1999-12-15 17:37:47.000	NULL		PR_MM	1	P	NULL	personal identifier	<unimeta><U_FLABS>PR_ID</U_FLABS> <UVALKEY>T</UVA...

Figure 4.5: Student Data Entities ASR_DET and PR_MM - Primary Key Result-set

4.3 Using Relational Algebra to Represent the ER Model

The ER model can also be represented using relational algebra which is the base foundation for relational databases, as shown in Elmasri and Navathe [226].

The data structures embedded in the forms contained in the XML snippet above, can be

represented in relational algebra as set out below:

$$\Pi_{(PR_ID,ASR_KEY_l,firstname,surname)}$$
$$(\sigma_{PR_ID.pr_id=ASR_DET.pr_id}(PR_M M \bowtie PR_ID))$$

The data model was transformed from ER to NoSQL as described in section 3.2 and the resulting formula should be equivalent to the following:

$$\Pi_{(}Student.PR_ID.pr_id, Student.PR_MM.ASR_KEY_l,$$
$$Student.PR_MM.firstname, Student.PR_MM.surname)(\sigma_{Student})$$

The new representation for the transforming the structure into an algebraeic formula, was based on Hidders et al.'s approach [227] and provides the following JSON representation as in figure 4.6 (JSON - Transformed Query) below:

▼ object {3}
DocumentID : 874a0b16-dd3b-4c23-8718-cb3e813d81e5
DocumentMetadata : Database specific data
▼ Student {2}
▼ PR_MM {2}
PR_MM :
ASR_KEY
▼ PR_ID {4}
PR_MM :
ASR_KEY_l :
Forename : first name
Surname : last name

Figure 4.6: JSON - Transformed Query

Then the JSON transaction was inserted into a noSQL database, see figure 4.7 (JSON Syntax).

```
{
 "DocumentID": "874a0b16-dd3b-4c23-8718-cb3e813d81e5",
 "DocumentMetadata":"Database specific data",
 "Student": {
  "PR_MM": {
   "PR_MM": "",
   "ASR_KEY":
  },
  "PR_ID": {
   "PR_MM": "",
   "ASR_KEY_I":
   "Forename": "first name",
   "Surname": "last name"
  }
 }
}
```

Figure 4.7: JSON Syntax

4.4 Parsing Uniface Data and Meta-Data

The following sections (4.4.1, 4.4.2) discuss two parsing techniques (for HTML and XML tools respectively) that were found to be unsuccessful when parsing Uniface XML-like code. Section 4.5 then describes in detail one novel technique (EDOM) which was found to be successful in doing so.

4.4.1 Can HTML Tools parse Uniface XML-like Code?

HTML documents are a special case of XML documents which start with an "<HTML>" tag and end with an "</HTML>" tag. All other HTML elements are nested within the root HTML element.

Existing HTML rendering tools are designed to tolerate inconsistencies in HTML documents. HTML interpreters are designed to detect unclosed tags. Some tools add the closing tag automatically and ignore the absence of root nodes. In contrast, XML interpreters are very strict and do not tolerate any inconsistencies.

4.4.2 Can XML Tools Parse Uniface XML-like Code?

In XML the term *markup* is used to determine nodes, special type of tags, which start with an opening bracket, which is a less than symbol, and a closing bracket, which is a greater than symbol. The element consisting of the less than symbol, is followed by the content and then a greater than symbol and is referred to as an XML tag.

Each XML document must be formatted in such a way that tags are nested inside the root tag which indicates the beginning and ending of the document. If the document satisfies this requirement, then the document is flagged as being a well-formed document. Uniface exported XML-like code does not always conform to this.

Parsing Uniface exported XML-like code is not as straightforward as parsing standard XML syntax because of the different technologies that XML tools use. Attempts to parse Uniface exported XML data resulted in errors due to ambiguity and the presence of Unicode special characters which Uniface reserves for internal purposes.

In many cases special characters which are typed within the content of the node contain ">" and "<" standing for the greater than and less than character. This deceives the parser which assumes it is beginning a new node or closing the current one (respectively). However, Uniface generates these characters for internal purposes only.

Therefore, it proved necessary for the author to build an XML interpreter using EDOM. This newly created tool, could either parse the file as a string or use an advanced reader with functionality equivalent to JSON.stringify() on the XML rather than manipulating it as a DOM object. This is discussed in the next section.

4.4.3 Can EDOM Tools Parse Uniface XML-like Code?

The following section 4.5, illustrates a successful novel technique, (using EDOM to parse a 4GL), that streams the XML file as a string and deals with the Uniface exported XML-like code format incompatibilities.

4.5 Encapsulated Document Object Model (EDOM)

4.5.1 Introduction to EDOM

The technique named Encapsulated Document Object Model (EDOM) was created by the company "First Object" but it is not a *de facto* standard in terms of parsing XML documents.

EDOM provides:

- a mechanism to parse a document,
- procedures to manipulate the DOM object which is contained within EDOM, and
- the method to simplify XML processing

EDOM works by setting up an index array for the document when the document is fully loaded, using the methods: "SetDoc" and "Load". At this stage the parser does not validate the document against well-formed documents. Also, it does not check against any DTD or schema which might be defined in the document.

This functionality relieves the developer of the complexity involved in manipulating any or all of: Document Object Model (DOM), Simple API for XML (SAX), Document Type Definition (DTD), Schema and Extensible Stylesheet Language (XSL) manipulation.

In order to avoid stack overflow issues for devices which have limited resources EDOM has also replaced the traditional recursion algorithm with the ElemStack algorithm which behaves like a memory stack, although the content is still stored in the memory heap.

The EDOM technique is built for devices with limited resources and also for huge files which consume the host's hardware. Therefore, EDOM provides dedicated functionality for parsing huge files efficiently.

4.5.2 Using EDOM for Large Files

The 'CMarkup' tool implements both ElmStack and also huge file handling algorithms, to serialise the content of the input file. The author used 'open' software implementation with a special directive 'CMarkup::MDF_READFILE', and 'FindElem' to find the next instance of a node. The following example demonstrates how a large XML file ("filename.xml") can be opened for parsing.

```
1 CMarkup parser;
2 parser.Open( "fileName.xml", CMarkup::MDF_READFILE );
3 while ( parser.FindElem("//object") )
4 {
5   // Handle the node content.
6   // Each node can be parsed using x_ParseNode
7 }
```

4.5.3 Parser: Tags and Nodes

Firstly, the parser performs the most basic check, to determine the tag type using 'CMarkup.GetTagName'.

Tags

A tag is the element that exists between the tag opener (less than) and tag closer (greater than) including the first word in between. For example

```
1 <DAT name="FORMPIC" xml:space='preserve'>
2 .... additional content
3 </DAT>
```

In this example '<DAT>' is referred to as a tag.

A list of the unique tags found in Uniface XML-like file is given below:

- <UNIFACE>

- <TABLE>
- <DSC>
- <FLD>
- <OCC>
- <DAT>

Any of the above tags can have optional tag attributes, and these tag attributes can have attribute values.

Tag Attributes:

- 'name' and
- 'xml:space'.

Attribute Values:

- 'FORMPIC'
- 'preserve'.

Lone end tags (also termed self-closing tags) are tags that do not have content and can be denoted using a single element such as '<DAT/>' which is equivalent to '<DAT></DAT>'. Lone ending tags can have attributes and attributes values.

Nodes

A node is the combination of tag, attributes, attributes values, content, and the tag closer.

Nodes can be of the following types:

- Element
- Comment
- Processing Instruction
- Document Type
- CDATA Section
- Text
- Lone ending tags

Files

A file can consist of one or more nodes and will always end with an end-of-file (EOF) marker.

4.5.4 Parser Processing Logic

The parser proceeds through the text attempting to identify nodes (and for each their node-type) and tags.

If the character at the current position is ‘less than’, then the parser assumes that it is a tag and performs a check against the next character.

If the next character is a slash then it recognises it as a closing tag “/>”.

In nodes, an exclamation mark “!” in the text denotes comments or a CDATA section or a DOCTYPE.

Processing instruction (PI) element is denoted by question marks before the tag name and after the last value of attributes. After inspecting the node type, the parser aims to find the closing tag for the current node.

If the parser finds that

- the next character is other than a left angle bracket “<” or
- a tab or <CR> or
- white space “ ”,

it will treat the data as ‘noise’ and ignore it. It will keep scanning.

If a right angle bracket “>” (end of tag) is detected and the parser will scan for more nodes.

If End of File (EOF) is detected the parser stops.

If the current node is a valid node of one of the node types listed above, the scanner looks for a non-white space character to identify the current node type.

If the scanner reaches the end of the node, it is assumed to be a white space node.

4.5.5 Example of Parser Processing XML

To explain the process more clearly, an example is provided of parsing the following XML:

<?xml version =”1.0”? ><Uniface >Version Nine </Uniface >

Figure 4.8 numbers each character position in this line of XML for ease of reading and tracking character positions when examining the parser's behaviour.

When x_ParseNode is called first, it looks for the first character, which is less than in this case. This denotes an open tag. Because the next character is a question mark it denotes the operation as a Processing Instruction (PI). After the type of node is determined then the type of the closing tag can be determined. The scanner sweeps through the text until it finds a question mark ‘?’ followed by a greater than (a valid node) to close the node.

```
( Two rows of coulmn position counters)
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

  < ? x m l   v e r s i o n   = " 1 . 0 "?   > < U n i f a c e   > V e r s i o n   N i n e   < / U n i f a c e   >


( Two rows of coulmn position counters)
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7

  < ? x m l   v e r s i o n   = " 1 . 0 "?   > < U n i f

  ( Two rows of coulmn position counters)
  2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5
  8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4

  a c e   > V e r s i o n   N i n e   < / U n i f a c e
```

Figure 4.8: XML Characters Numbered by Column Position

When the function x_ParseNode is called next, it will scan the next character which is open bracket (less than symbol). As previously explained, the parser will denote this as an open tag indicator.

The next character ‘U’ is not a special character (such as a ‘?’ or ‘!’) and so the node is flagged as a normal XML node. The parser will keep looking for the closing tag (greater than symbol).

On the next call to the function, the parser will encounter an empty space (which it will ignore) and on the following call then the text “Version Nine” (which sets the current installed version of the Uniface IDE to be 9.*). Therefore, the parser will denote the current node as a text node and scan for the closing tag.

The next call to the function determines that the next character in the parsed text is a right angle bracket “>”. The right angle bracket was preceded by a slash “/” which denotes a closing node. The function will continue scanning and parsing until end of file (EOF) is encountered.

4.6 Why Unicode is Needed for the Uniface Character Set

Uniface's XML-like code contains characters outside the standard ASCII character set. Therefore Unicode was selected to correctly capture and interpret this extended character set.

To encode English language characters it would be sufficient to use the ASCII symbol set table, Paterson, ISO/IEC 8859-1: 1997 [228], due to the limited number of characters in the English language. However, other languages are not able to be interpreted solely using the ASCII table.

The Unicode Standard, Allen et al. [72], shows that Unicode uses a wider character set. This results in the need for more space required to store Unicode characters as opposed to ASCII ones. Further detailed comparison between Unicode and Non-Unicode characters is provided in table 4.2 below.

Comparison between Unicode and non-Unicode characters	
Unicode	**Non-Unicode**
DATA TYPES AVAILABLE	
nchar, nvarchar, ntext,	char, varchar, text.
SIZE OF FIELD v SIZE OF DATA	
The size of the field determines the size of data stored in it.	The size of the field determines the size of data stored in it.
NCHAR/CHAR DATA TYPES	
The nchar data type is stored based on the number of characters entered, where if the size of the entered data is less than the defined data type size then the remaining space will be padded with white spaces.	The char data type is stored based on the number of characters entered where if the size of the entered data is less than the defined data type size then the remaining space will be padded with white spaces.
NVARCHAR/VARCHAR DATA TYPES	
nvarchar is stored in the same the way that nchar is stored. However, padding does not apply to it.	varchar is stored in the same way char is stored. However, padding does not apply to it.
BIT LENGTH per CHAR	
Requires between 8 and 32 bits per character.	Requires 8 bits storage.
MAX LENGTH FOR VARCHAR	
The maximum length is 4000 characters.	The maximum length is 8000 characters.
USAGE	
Usually used with systems that manage international localisation or special cases like Uniface technique in injecting characters which have specific meaning without confusing them with developer code.	Used usually for mono-language systems which are based predominantly on the English language.

Table 4.2: Comparison between Unicode and non-Unicode Characters

Unicode characters are stored using double-byte character sets (DBCS) which means that storing a Unicode character will require double the space of that required to store a non-Unicode character.

This in turn means that to parse the content of Uniface files, a Unicode compatible parser is required.

In practice, of course, not every database engine supports Unicode characters. However, modern database engines do support Unicode but each uses a different mechanism to resolve any problems while retrieving data containing international text characters, Hsing and Yaung [229].

4.7 Conclusion

This chapter has covered the first stage of the implementation of the author's project.

Cleaned XML-like data was extracted from Uniface from which an ER models was extracted. It could also be modelled using Relational Algebra.

The Uniface data was successfully parsed in the form of EDOM structures which ensured that valid nodes and syntax could be extracted.

Next, chapter 5 Implementation II - Reverse Engineering Uniface Schema, looks in more detail in terms of reverse engineering - that is extracting - Uniface Schema using either a Document Type Definition (DTD) approach and also using an XML Schema Definition (XSD) approach.

Chapter 5

Implementation II - Reverse Engineering Uniface Schema

Significant sections of this chapter have been adapted from or have appeared in the author's published paper Yafi and Fatima [9].

The previous chapter, Implementation I - Extraction, ER and Relational Algebra Modelling - (chapter 4) covered extracting XML from Uniface, cleaning it, and deriving data models from that in both Entity Relationship (ER) and Relational Algebra formats.

It also covered the EDOM structure used to hold such data.

This chapter is devoted to the task of Reverse Engineering Uniface Scheme in both Document Type Definition (DTD) form (section 5.1) and also as XML Schema Definition (XSD) form (section 5.2), where a higher level of schema information is provided than using the DTD format.

5.1 Reverse Engineering Uniface Schema Using Document Type Definition (DTD) Form

This is the first of two possible approaches to validate the cleaned XML as output from Uniface - to check that Uniface syntax is precisely applied to the input.

The second approach - using XSD - is discussed in section 5.2.

5.1.1 Introduction to DTD Schema

The term Document Type Definition (DTD) is commonly used to describe an XML document's meta-data. It can also be used to validate the content of XML documents against the vocabulary

used within it.

When using DTD it becomes possible to perform grammar checking and syntax validation, which is of particular relevance for the current Research.

The cleaned and parsed XML with Encapsulated Document Object Model (EDOM) section 4.5, needs to be both well-formed and valid - both terms are discussed below.

Well-formed XML Documents

- Each node must have opening and closing tags if the node contains character data where character data are the series of characters in the node. For example, character data in figure 4.8 are the range from position 33 to position 54, including the white-space at position 54,
- Nodes that do not have character data can either have opening and closing tags or can formatted as Lone end tags as described in section 4.5.3,
- Attribute values are optional,
- Attributes values must be surrounded by quotes, either single or double (but matching),
- Nodes can be nested where the nested nodes must have their opening and closing tags within the parent node.

Valid XML Documents

Valid XML documents are documents that are composed from DTDs.

Components of an XML Document Figure 5.1, taken from Yafi and Fatima [9], provides a high level schematic overview of the main component parts of an XML documents within an XML interpreter.

As can be seen from figure 5.1, it shows that within an XML document there is content (in XML), the presentation format of that content (in XLS), and the XML document's structure (set out as XSD).

DTDs

Within a valid XML file, a DTD could be embedded within the XML document (Internal DTD) or in a separate file with a reference to it in the XML content (External DTD). The DTD not only defines the elements that could be written in the XML document but also the order of these elements and associated attributes. It can define attributes as mandatory or optional.

The concept of a DTD predates XML name-spacing (Bray et al. [230]), but supports string data-types only. As part of the author's research, the aim is to extract the DTD of the cleaned XML file from chapter 4. Reverse engineering the XML file to find its DTD can cover these purposes:

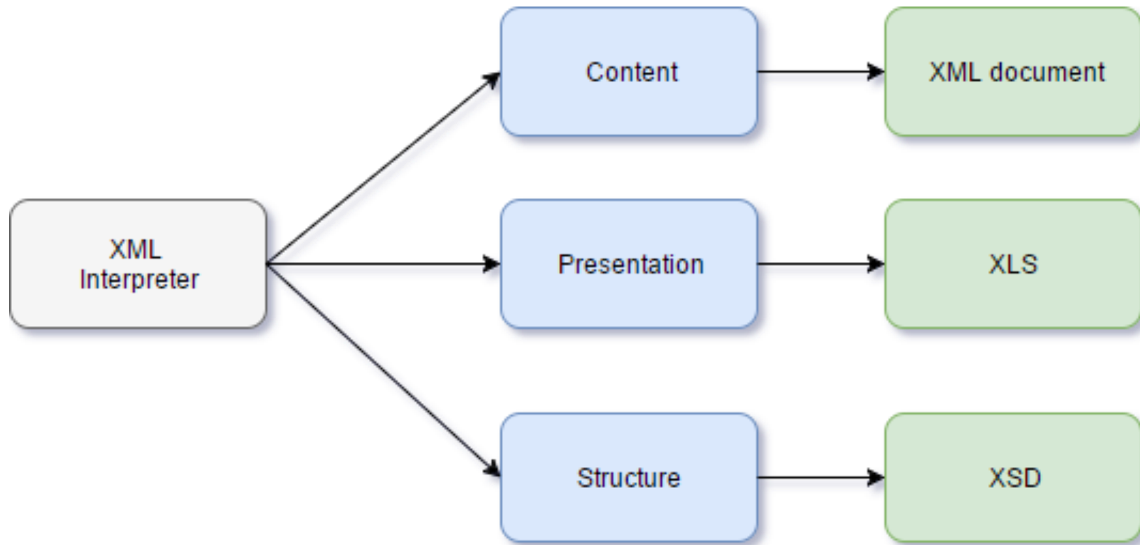

Figure 5.1: High Level Schematic of XML Interpreters. Taken from Yafi and Fatima [9]

- To validate the existing XML file or its smaller parts as per section 6.1 Splitting up Large XML Files - Extracting TABLE Elements,
- Parse XML correctly with the means of existing or bespoke tools,
- To generate sample XML files based on the DTD for testing purposes.

5.1.2 Uniface Document Type Definition Syntax

Uniface DTD It is important to understand DTD syntax because Uniface exports XML files without the associated DTD. However, the Uniface exported XML-like code contains a reference to the Uniface-specific DTD file internally, as follows:

```
1 <!DOCTYPE UNIFACE PUBLIC "UNIFACE.DTD" "UNIFACE.DTD">
```

The presence of such a reference to the former existence of a DTD file, implies that it should be possible to recreate a DTD from the processed XML.

Since the original Uniface DTD is not available, there is a need to re-create the matching schema and DTD view of the Uniface XML. Figure 5.2 (taken from Yafi and Fatima [9]) represents the Uniface DTD output as reverse engineered by the author.

DTD Syntax

The DTD directive starts with <!DOCTYPE then a list of entities which are expected next.

DTD supports the (#PCDATA) data type which represents a string of a finite number of characters. A DTD can have both internal Type Identifiers and external ones).

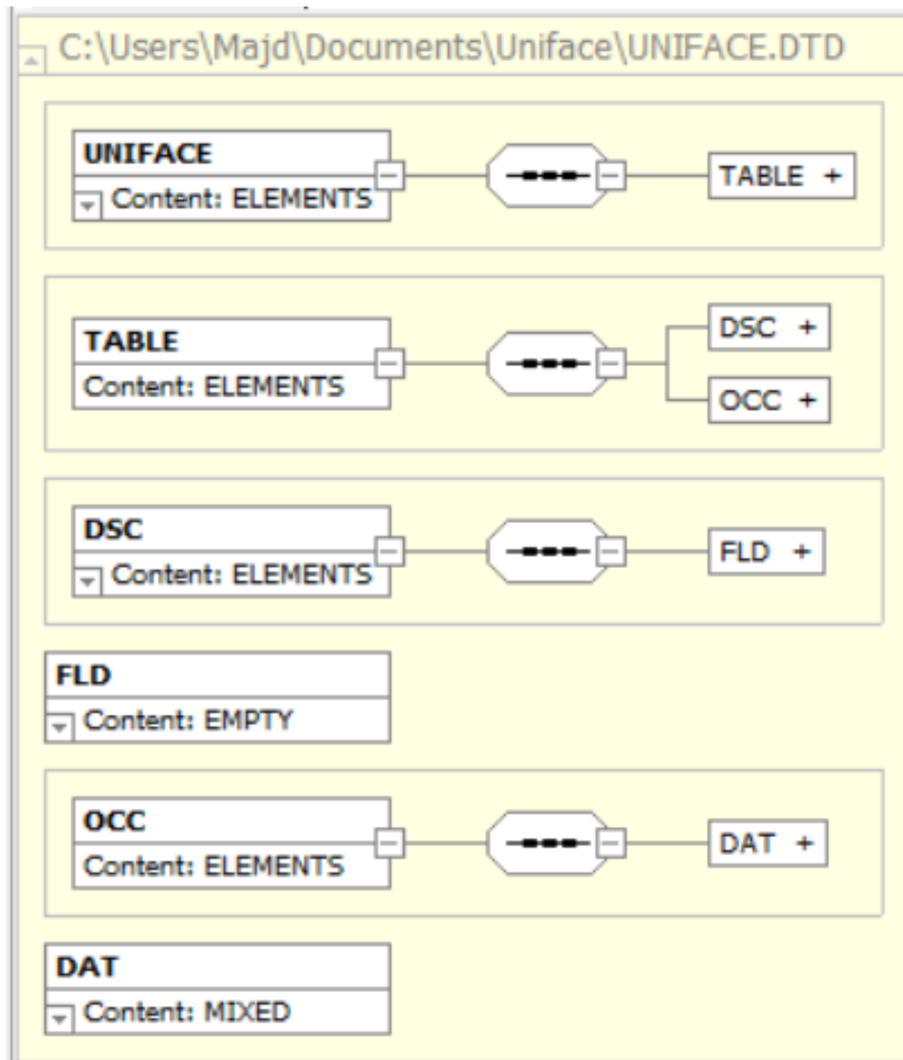

Figure 5.2: Uniface Schema in Document Type Definition (DTD) Format. Taken from Yafi and Fatima [9]

Elements

The body contains a list of elements where each element starts with <!ELEMENT and then provides the element name and data type. It is also possible to create an object of multiple elements.

For example: Uniface as an element in figure 5.2 is formed of TABLE and DSC.

In the example in figure 5.2, UNIFACE is an internal identifier, consisting of TABLE and DSC combined in one XML file.

Attributes and Entities

In addition to elements, a DTD enables the definition of Attributes and Entities.

Attributes are added before the closing tag using the attribute name, equals sign, and then the attribute value surrounded by quotes.

Attributes can be String, Tokenised or Enumerated. Attribute types can be explicitly defined as one of the following items in the list:

- CDATA (Character data, String attribute type - a series of characters),
- ENTITY (Represents an external entity, Tokenised attribute type),
- ENTITIES (A list of external entities, Tokenised attribute type),
- ENUMERATION (List of value, Enumerated attribute type),
- ID (Unique identifier, Tokenised attribute type),
- IDREF (Reference to another ID, Tokenised attribute type),
- IDREFS (Reference to multiple IDs, Tokenised attribute type),
- NMTOKEN (CDATA for a XML name, Tokenised attribute type),
- NMTOKENS (CDATA for XML names, Tokenised attribute type),
- NOTATION (An element can be referenced to the notation in the DTD, Enumerated attribute type).

Entities are placeholders and can be further characterised as:

- Built-in entities,
- Character entities,
- General entities,
- Parameter entities.

Entities can have indicators such as:

- + <ELEMENT element-name (child1+)>
- * <ELEMENT element-name (child1*)>
- ? <ELEMENT element-name (child1?)>
- , <ELEMENT element-name (child1, child2)>
- | <ELEMENT element-name (child1 | child2)>

Special Attribute Categories

There are also five other special attribute categories:

- Ampersand: ‘&’,
- Single quote: ‘'’,
- Greater than: ‘>’,
- Less than: ‘<’,
- Double quote: ‘"’.

Uniface adds ‘Golden characters’ with special meanings to the standard Unicode character set (as discussed in section 4.6). To deal with these, additional entries need to be defined in the DTD to cater for Unicode characters with special meanings in Uniface, within the XML file.

Reverse engineering the DTD of the processed XML file ensures that the resultant XML file it is both valid and well-formed. Therefore, the processed XML file is standard and can be reliably used with XML parsers and interpreters. This is a first step leading on to section 5.2 which essentially does the same task but outputs an ‘XSD’ XML Schema file which provides a richer set of information than that provided by the DTD approach.

5.2 Reverse Engineering Uniface Schema Using XML Schema Definition (XSD) Form

5.2.1 Introduction to XSD Schema

XML schemas are another way to validate XML and check that the Uniface syntax is precisely applied to the input. Likewise, just as a DTD can detect well-formed documents and a valid structure, XSD plays the same role but with more features.

For this author's research, it was thought highly desirable to develop an XSD schema to help understand the structure of a Uniface document and the vocabulary used to form it. A screen view of the scheme being the outcome of the extracted XSD file, is shown in figure 5.3, taken from Yafi and Fatima [9]. This XSD model validates the exported Uniface XML as well as the extracted XML files using an EDOM data structure model, which has been explained in section 4.5.

The main feature of XSD is that it uses XML syntax which makes it extensible and therefore potentially adaptable. In addition, it does not only recognise data types but also supports name-spaces which allow the schema to be organised into named logical fragments.

As one of the main features it is a standard representation for XML documents, Gao et al. [231].

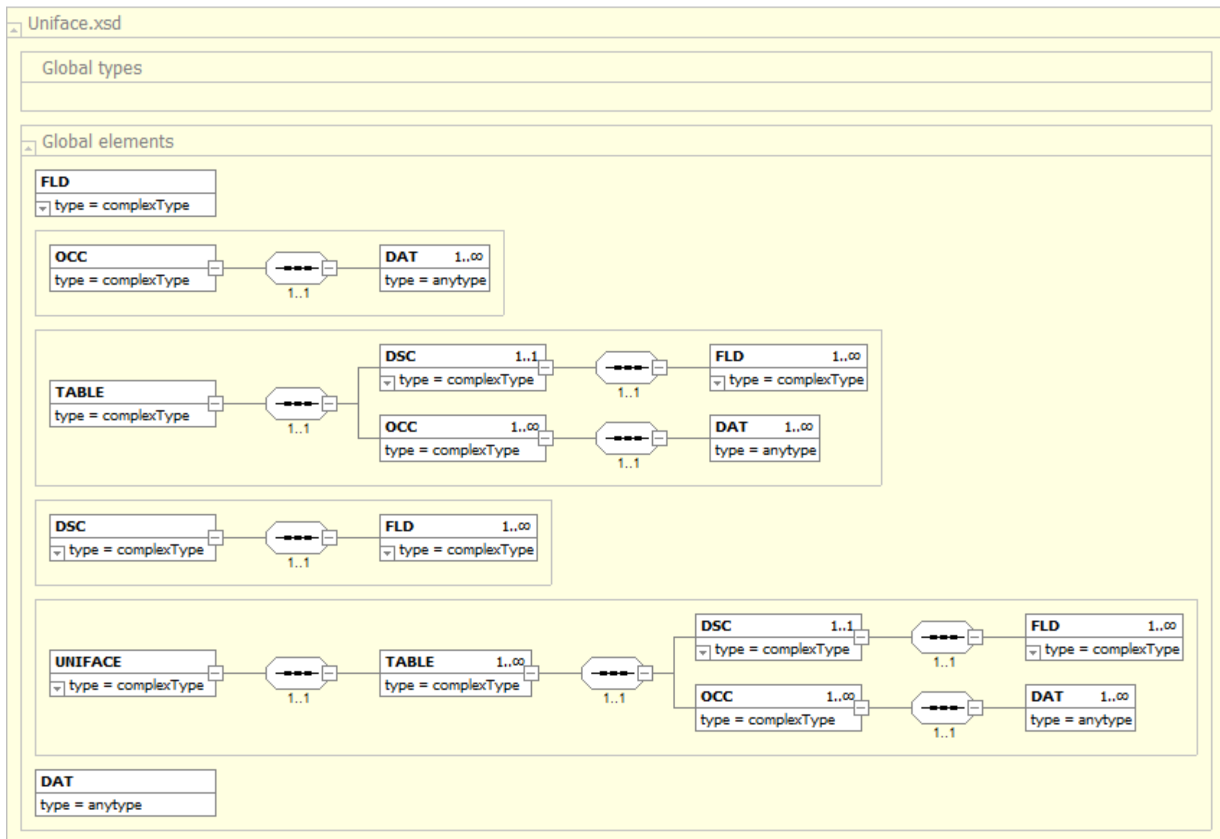

Figure 5.3: Uniface Schema in XML Schema Definition (XSD) Format. Taken from Yafi and Fatima [9]

5.2.2 XML Schema Definition Syntax

The XSD is another XML file (a schema file) that has to be stored separately from the original XML file that it represents. The rationale behind using schema files is that it resolves any ambiguity which might occur when parsing the original Uniface-exported XML file. There should be a unique interpretation for the parser and it has to know whether the targeted node is an XML node or a meta-data node. These were previously discussed in Section 4.5.3.

A XML Schema Instance (XSI) is used to provide additional attributes to associate the schema file with the XML document. This could be, for instance, to direct the parser to locate a specific XSD file. In the XML code, the xsi:schemaLocation directive can be used (if a schema is present), followed by the XSD file location.

The next fragment illustrates the way a name-space can be used to instruct validating software, such as parsers to check against it.

```
1 xmlns = "http://www.Uniface.com"
```

5.2.2.1 Defining Elements in XSD - Simple Types

An element is the lowest level component of an XML file.

This is an example of how creating an element with type 'string' is shown in the XML of a XSD file.

```
1 <xs:element name = "FLD" type = "xs:string"/>
```

Additional attributes can provide specific restrictions or for more accuracy.

For example, to limit FLD in the previous example to be 20 characters long then fixed="20" needs to be added. The new fragment would then read:

```
1 <xs:element name = "FLD" type = "xs:string" fixed = "20"/>
```

Attributes such as *default values* and *required field flag* are also used to give meaning to an XML node.

In order to support continuous re-engineering of the system, it is a prudent step to simplify the XSD or DTD to tolerate unexpected errors. The disadvantage of doing so is that this approach might cause lack of accuracy when representing the targeted system.

To restrict XML elements to have specific values, one or more of the following attributes can be used to refine the content of the node:

- enumeration,
- fractionDigits,
- length,
- maxExclusive,
- maxInclusive,
- maxLength,
- minExclusive,
- minInclusive,
- minLength,
- pattern,
- totalDigits,
- whiteSpace.

5.2.2.2 Defining Elements in XSD - Complex Types

In contrast to DTDs, XSD also allows complex data-types to be created. In order to create complex types the root element should incorporate all other types within it and define a sequence of elements that should occur.

Valid items in the sequence are one or more of the following:

- Empty,
- Elements only,
- Text only,
- Mixed,
- Indicators,
- <Any>,
- <Any Attribute>.

Figure 5.4, (from XML Schema Types, w3.org, [10]), lists XSD simple data-types.

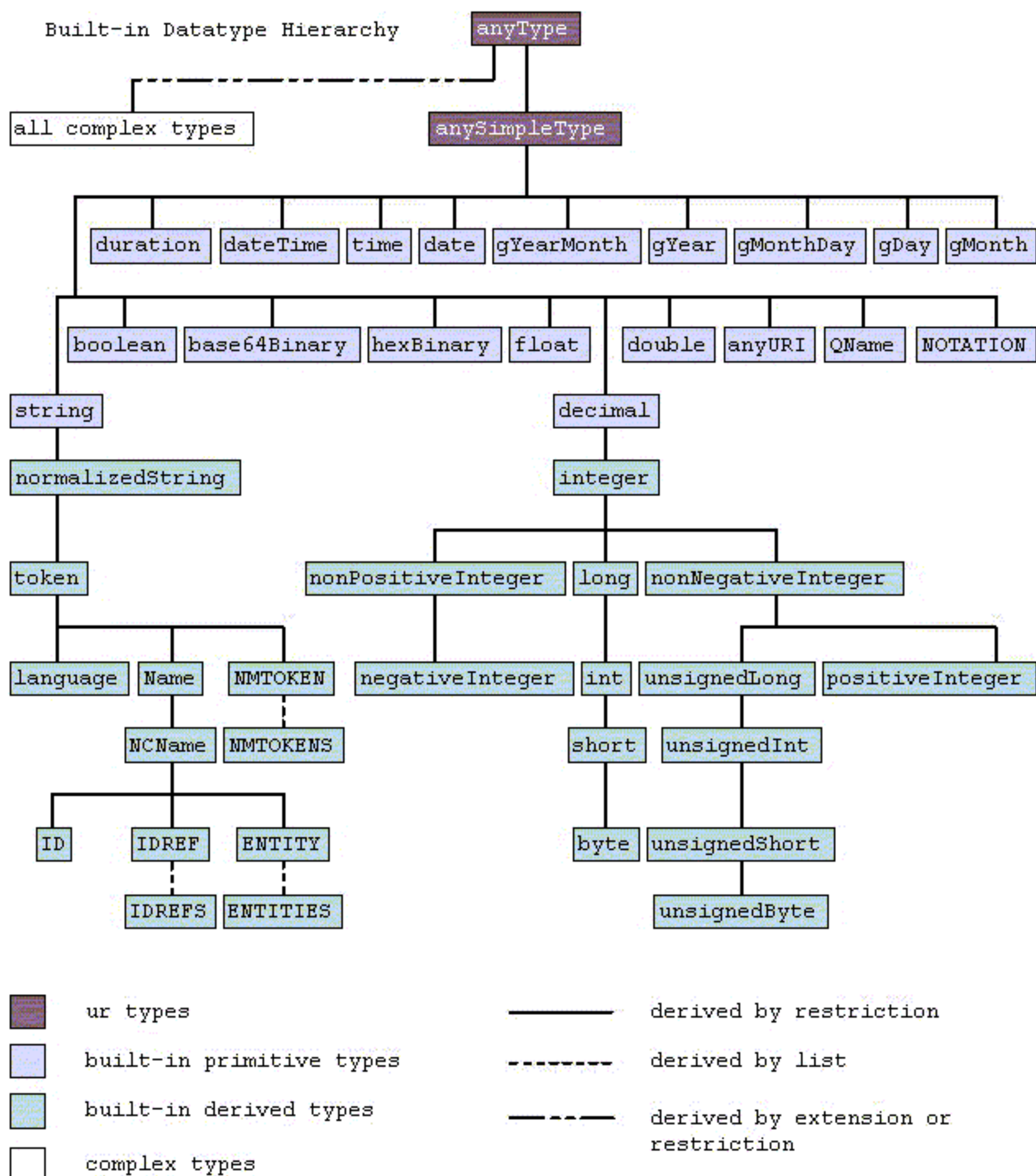

Figure 5.4: XSD Simple Data-Types Source: XML Schema Types, w3.org [10]

XSDs are a form of advanced DTDs where XSDs are extensible, support data-types and default values, are written in XML, and can reference external schemas. These advantages enable

mirroring XSD data-types to the selected language for traspiling (C#). The implementation is provided in appendix C (C# Code to Transpile Uniface into C#).

5.3 Conclusion

This chapter has dealt with using firstly Document Type Definitions (DTD) and secondly XML Schema Definitions (XSD) as possible ways to reverse engineer and thus validate the XML and its embedded data structures, as output from Uniface.

It also discussed the advantages of using the XSD method since it provided a richer and deeper picture of those data structures, and thus potentially more accurate validation, despite the possibility that some of these additional checks might be inappropriate.

The XSD method was chosen for this research since it provided fuller information.

The next step is to extract Uniface Elements, which chapter 6 discusses.

Chapter 6

Implementation III - Extracting Uniface Elements

This chapter covers the key processes involved in extracting Uniface elements namely:-

- Splitting up Large XML Files - Extracting TABLE Elements (section 6.1),
- Exporting XML to SQL (section 6.2),

together with a discussion of

- Parsing Techniques for Programming Languages (section 6.3), and
- Meta-syntax Notation (section 6.4).

6.1 Splitting up Large XML Files - Extracting TABLE Elements

6.1.1 Need for Dividing a Large XML File

Uniface exports the code-base in XML-like format as described in section 4.2.1. Nested nodes contain segments of Uniface code and meta-data. In the earlier sections (5.1) have shown the basics of parsing an XML structure and applied that knowledge to a Uniface file. This is applying the principles that the XML file should be both valid and well-formed (as previously discussed in section 5.1.1). In the same section it has also been shown how to manipulate unknown characters and interpret them correctly.

It was found that a reliable approach was to create smaller files which contained less code in order to narrow down any possibly faulty code to a minimum. In order to fragment the code-base

DTD and XSD models were found to aid in deciding where to divide the code file in smaller parts. DTD and XSD were illustrated in figures 5.2 and 5.3 and discussed in sections 5.1 and 5.2 respectively.

It was therefore decided to split the XML over smaller files based on size of data structures. This approach has drawbacks, because a Uniface XML file contains code and interrupting it at any position means that code fragments might become fully or partially truncated. Therefore the XSD schema has been used to split the large XML file at only TABLE closing elements.

6.1.2 Splitting the XML File

To split the file, first the nested elements needed to be extracted and written in the same sequence to an external file. This operation was very expensive in term of resource consumption.

An algorithm using EDOM (explained in section 4.5) succeeded.

The outputs from the algorithm set out below, are explained immediately afterwards.

```
1  split_Uniface()
2  {
3    int nObjectCount = 0, nFileCount = 0;
4    CMarkup xmlInput, xmlOutput;
5    xmlInput.Open( "Uniface.xml", MDF_READFILE );
6    xmlInput.FindElem(); // root
7    str sRootTag = xmlInput.GetTagName();
8    xmlInput.IntoElem();
9    while ( xmlInput.FindElem() )
10   {
11     if ( nObjectCount == 0 )
12     {
13       ++nFileCount;
14       xmlOutput.Open( "piece" + nFileCount + ".xml", MDF_WRITEFILE );
15       xmlOutput.AddElem( sRootTag );
16       xmlOutput.IntoElem();
17     }
18     xmlOutput.AddSubDoc( xmlInput.GetSubDoc() );
19     ++nObjectCount;
20     if ( nObjectCount == 1 )
21     {
22       xmlOutput.Close();
23       nObjectCount = 0;
24     }
```

```
  }
  if ( nObjectCount )
    xmlOutput.Close();
  xmlInput.Close();
  return nFileCount;
}
```

```
File  Edit  View  Tools  Window  Help

split_xml_15GB()
{
  int nObjectCount = 0, nFileCount = 0;
  CMarkup xmlInput, xmlOutput;
  xmlInput.Open( "Uniface.xml", MDF_READFILE );
  xmlInput.FindElem(); // root
  str sRootTag = xmlInput.GetTagName();
  xmlInput.IntoElem();
  while ( xmlInput.FindElem() )
  {
    if ( nObjectCount == 0 )
    {
      ++nFileCount;

4517
```

Figure 6.1: XML File Splitter: Code and Output

The output of the code in figure 6.1, is an integer (in this example 4517) representing the number of newly-split files created.

These files are contained in a child directory.

The first file holds meta-data which is used later for different purposes. Every file apart from the first file, contains TABLE elements (an example of such a TABLE is provided in appendix G). Figure 6.2 below, shows a small extract of the XML to demonstrate that the hierarchy TABLE -> OCC -> DAT has been successfully extracted. Further explanation has been provided in section 6.1.3 below.

```
TABLE
  OCC
    DAT 2013-04-02T10:56:32.99
      name = UTIMESTAMP
    DAT 2013-04-02T11:46:41.08
    DAT AAAAERPS
    DAT Q
    DAT Process Resit Flag (SEPT)
    DAT 0
    DAT 0
    DAT 6
    DAT 47
    DAT N
    DAT N
    DAT F
    DAT GENLIB
    DAT 0
    DAT  #Comment -------------------------------------------  #Comment start_of_references  #ifdefined TEMPLATENAME  #info symbol
    DAT Params  string ASRKEY : IN  string SERSQ : IN    Endparams    clear/e '[redacted]'  ASR_KEY.[redacted]= ASRKEY  SER_SQ.[redacted] = SE
    DAT clear
    DAT retrieve  if ($status <0)    message $text(1762) ;error  endif
    DAT retrieve  if ($status <0)    message $text(1765) ;error  endif
    DAT store  if ($status <0)    message $text(1500) ;error    rollback  else    if ($status = 1)      message $text(1723) ;no ch
    DAT erase  if ($status <0)    message $text(1763) ;error    rollback  else    if ($status = 1)    message $text(1634) ;not a
    DAT entry LRESIT  if (RESIT_FLAG.CER_DET >= "A")  if ([redacted] = "" & [redacted] = "" %\    & [redacted]
    DAT         &uFRM;TYP=E&uSEP;NAM=[redacted]&uSEP;WID=13&uSEP;HEI=1&uSEP;HOC=13&uSEP;VOC=1&uFRM;    &uFRM;TYP=E&uSEP;NAM=C
    DAT #DEF
    DAT Process Resit Flag (SEPT)
```

Figure 6.2: Uniface TABLE Element Structure

6.1.3 Extracting 'OCC' Elements and Finding TABLE Elements

6.1.3.1 Extracting 'OCC' Elements

The term 'OCC' is an Uniface specific term which is not a documented term. However, based on the unearthed XSD specification as in figure 5.3, it was found that Uniface stored snippets of code within '<DAT>' nodes which are themselves inside '<OCC>' nodes. The description 'OCC' is a label assigned by the Uniface system. Its name is meaningless but its presence carries meaning as a constituent part of the schema. Its existence can be seen in figure 5.2 and figure 5.4.

Therefore, an algorithm to reveal the content of these nodes was needed, as the output from this algorithm will become the input for the translation engine which will be dealt with in chapter 8.

The '<OCC>' nodes were selected for decomposition in the following example, since it could be seen from the DTD or XSD that '<OCC>' contained '<DAT>' nodes.

The algorithm listed below, traverses '<OCC>' nodes as previously identified, using DTD or XSD schema information in their output files (see figures 5.2 and 5.3). Then the routine exports the nodes one by one to a new, valid and well-formed XML file.

```
1 split_TABLE()
2 {
3   int nObjectCount = 0, nFileCount = 0;
4   CMarkup xmlInput, xmlOutput;
5   xmlInput.Open( "piece1.xml", MDF_READFILE );
```

```
6   xmlInput.FindElem("/UNIFACE/*");
7   str sRootTag = xmlInput.GetTagName();
8   xmlInput.IntoElem();
9   while ( xmlInput.FindElem() )
10  {
11    if ( nObjectCount == 0 )
12    {
13      ++nFileCount;
14      xmlOutput.Open( "nodes/" + nFileCount + ".xml", MDF_WRITEFILE );
15      xmlOutput.AddElem( sRootTag );
16      xmlOutput.IntoElem();
17    }
18    xmlOutput.AddSubDoc( xmlInput.GetSubDoc() );
19    ++nObjectCount;
20    if ( nObjectCount == 1 )
21    {
22      xmlOutput.Close();
23      nObjectCount = 0;
24    }
25  }
26  if ( nObjectCount )
27    xmlOutput.Close();
28  xmlInput.Close();
29  return nFileCount;
30 }
```

6.1.3.2 Finding ‘TABLE’ Elements

Finding a specific element requires searching the content of the file using the *xPath* expression. XPath is a language that is written to return XML nodes from hierarchy tree, Berglund et al. [232], Clark and DeRose [233].

In an example run produced by the researcher, 5518 sub-nodes were found which contained Uniface code.

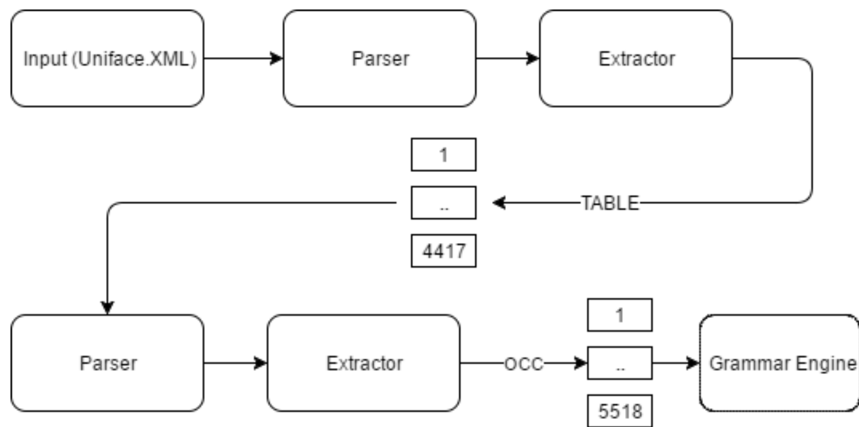

Figure 6.3: The Process of Extracting TABLE and 'OCC' Elements Taken from Yafi and Fatima [9]

Figure 6.3 taken from Yafi and Fatima [9], illustrates the procedure to extract Uniface code from the code base, that includes data and meta-data, to the content of '<OCC>' elements, which contain pure code but are still encoded.

Before fetching the code into a grammar engine the code needs to be cleaned to improve it for parsing readability purposes.

To illustrate this cleaning process, the following snippet of code shows Uniface internal comments.

```
#Comment

#Comment end_of_references


#Comment ——————————————————————————
#Comment start_of_mappings
#Comment end_of_mappings


#Comment ——————————————————————————
#Comment start_of_symbols
#Comment end_of_symbols


#Comment ——————————————————————————
```

These comments are not typical development comments. Therefore, each line of them can be

safely deleted. It is believed that Uniface uses these lines as directives to drive the IDE behaviour. Clones of this snippet were seen repeatedly in the exported code-base.

6.2 Exporting XML to SQL

Transforming Uniface XML into SQL relational databases can simplify the task of searching data and referencing code blocks. Data held within XML is generally and naturally represented using a tree structure, which enables the transformation to occur. However, reading and parsing Uniface XML-like has been a challenge, and proved to be an unsuccessful research avenue.

Different techniques have been used so far, one of which, using XPath, Suri and Sharma [234] was examined more closely. However, due to the limits of SQL capabilities and the usage of encoded characters in the source files, it became easier not to take the route of targeting XML input as a finite number of characters than using SQL bulk insert to fetch the nodes into database tables using XPath, and then querying them.

Another difficulty was the nature of SQL's row limit.

By examining the Uniface SQL Database it has been found that it uses overflow tables due to the limit of how many bytes one SQL row can have according to Rankins et al. [235]. Recombining rows to form complete statements is very complex to engineer. This approach is unlikely to succeed, because there is no clear and unambiguous method to identify the link between the starter row and any overflow rows. Larson et al. [236] have described some approaches and enhancements to circumvent the storage limit.

The difficulties described above meant that the approach of storing the XML in a relational database, with the benefits of querying it using SQL is not viable, so the research turned to parsing the XML from XML fragments instead.

6.3 Parsing Techniques for Programming Languages

To construct, validate and parse grammar for XML there are both heuristic methods and also formal ones which may be used to eliminate ambiguity as shown in Fan et al. [237], Murata et al. [238] whereas other researchers, Handoko and Getta [239] aimed to use XML algebra for coding a formal syntax of XML elements. Henglein and Rasmussen [240] used the Progressive Tabular Parsing method.

Grammar recovery can be processed manually, or with semi-automated tools or can even be fully automated according to Duffy and Malloy [241]. Other researchers, such as Mooij et al. [242] used open source platforms to recover the legacy design and generate a modern equivalent one. Parsing expression's grammar and Backus–Naur form techniques Gabriëls, et al. [243] were used when parsing functional languages such as F#, Kubicek et al. [244].

XML is commonly represented using a tree structure, given the fact that XML is built using nodes and sub-nodes. However, other researchers, have presented their XML documents using Graph representation, such as Pokorný [245], Hristidis et al. [246], Ceri et al. [247], Deutsch et al., 1999 [248] and Deutsch et al., 1998 [249].

6.3.1 Parsing Using Regular Tree Grammar (RTG)

Regular Tree Grammar (RTG) as described by Brainerd [250], Nishida and Maeda [251], and Teichmann et al. [252] use a formal method to depict discrete trees using a single path tree structure.

RTG is defined as:

$$G = (N, \Sigma, Z, P)$$

where

- N is a finite set (non-terminals),
- Σ is an alphabet with $N \cap \Sigma \neq \emptyset$ (terminals),
- $Z \in N$ (initial non-terminal), and
- $P \subseteq N \times (N \cup \Sigma)$. P is a set of productions of the form $A \to t$ where $A \in N$ and $t \in T_\Sigma(N)$ where $T_\Sigma(N)$ is the set of trees composed from symbols in $\Sigma \cup N$ according to their arities given that non-terminals are nullary.

An example of the production rules P where E is a non-terminal and i is a terminal:
$S \to E$
$E \to (E + E)$
$E \to i$

6.3.2 Parser Algorithms

According to Parr and Fisher [130] the problem of how a parser is constructed has not yet been solved, although powerful methods exist such as Parser Expression Grammars (PEGs) according to Medeiros et al. [253] and Ford [254]. Some use LL Parsing grammar which is leftmost derived parsing. $LL(K)$ denotes Left parsing for K tokens. Terry [255] stressed that the value of using this model lies in its simplicity of implementation.

Using the same expressions (E being non-terminal and i being terminal) then applying $LL(k)$ on $w = ((i + i) + i)$ results in:
$S \Rightarrow E \Rightarrow (E + E) \Rightarrow ...$
The ambiguity that is present in the last expression can be interpreted as one of the following rules:
$E \to (E + E)$

$E \to i$

Ideally parsers should not need to backtrack to determine the next statement.

6.3.3 Using LL(k) Parsers

This research aimed to use the LL(k) technique to parse 4GL syntax. The tool set which was designed to assist researchers was the Purdue Compiler Construction Tool Set (PCCTS), Parr et al. [81]. This tool was covered in section 2.2.3 earlier.

6.4 Meta-syntax Notation

Meta-syntax notation is the set of terms that can informally describe natural languages or can be based on formal rules to describe computer languages.

This research utilised the formal meta-syntax notation named EBNF to describe the grammar rules of targeted languages.

6.4.1 Meta-syntax Notations as Formal Input

Extended Backus-Naur form notation, Wirth [256], is considered later, in section 8.3.2, for context-free grammar syntax abstraction in combination with Abstract Syntax Trees (AST) Jones [257]. An AST is generated from grammar rules and input syntax. In AST, nodes are labelled and named like trees (child nodes).

AST were discussed previously in section 3.4.3 and illustrated in figure 3.4 Parser Structure: Lexer, Tokeniser, and Parse Tree (Author's Diagram).

6.4.2 Using SQL 'CREATE TABLE' as an Example

SQL is itself a 4GL language, Kipp [45], Harrison et al. [79]. Using AST and LL(k) approaches to SQL, it can be shown that *Create Table* statement grammar can be defined as in the listing below.

```
1 create_table_stmt
2  : K_CREATE ( K_TEMP | K_TEMPORARY )? K_TABLE ( K_IF K_NOT K_EXISTS )?
3    ( database_name '.' )? table_name
4    ( '(' column_def ( ',' column_def )* ( ',' table_constraint )* ')' (
   ↪  K_WITHOUT IDENTIFIER )?
5    | K_AS select_stmt ) ;
```

The grammar can be visualised and transformed into a tree structure. Figure 6.4 shows a truncated output due to the large number of elements included. However, the visible segments

reflect the grammar and illustrate it with a Rail-road/Syntax diagram, as described in Brisson's US Patent [258].

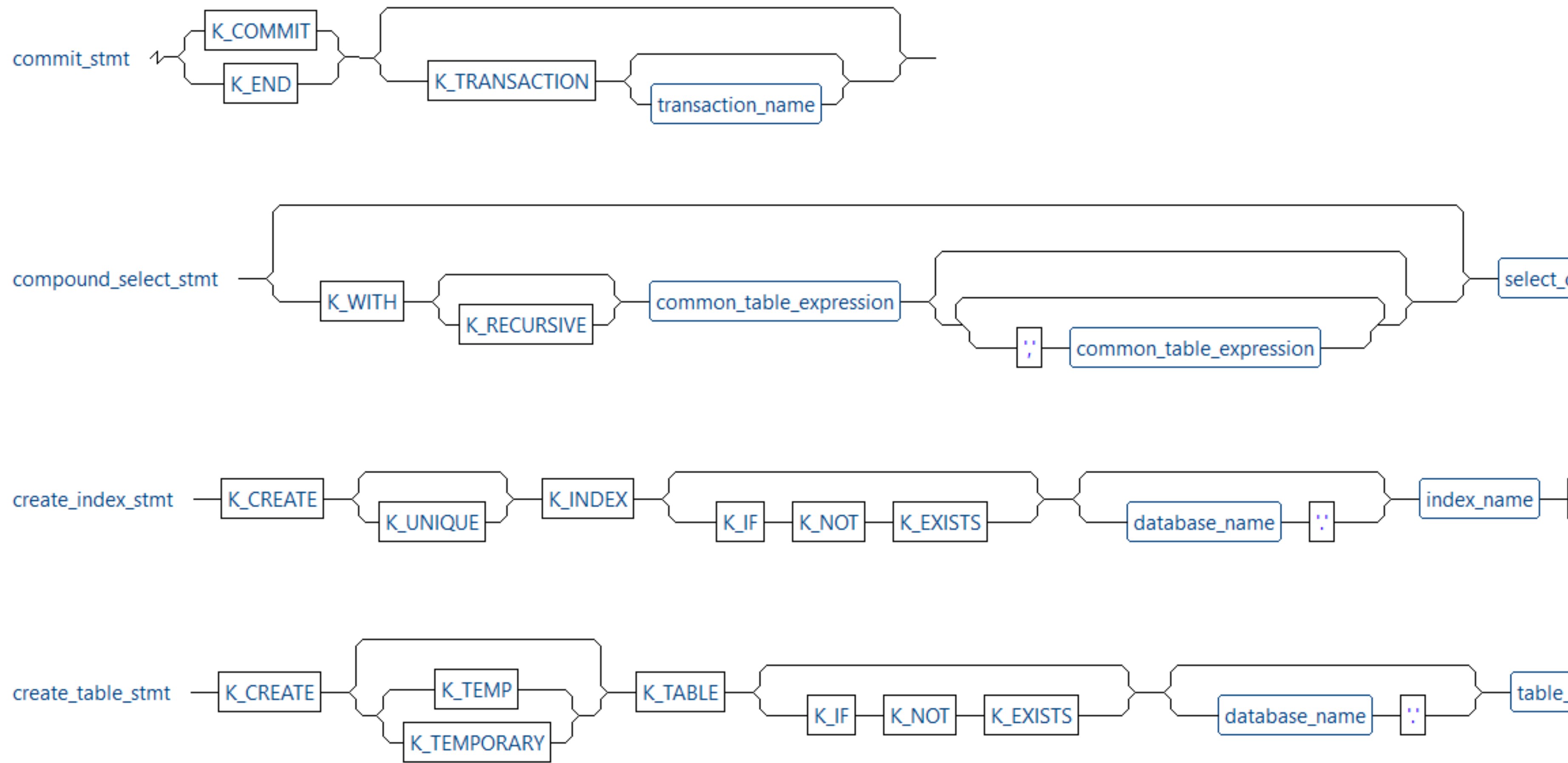

Figure 6.4: SQL Grammar - Create Table

6.4.3 Example: Parsing SQL 'CREATE TABLE'

For instance, the following code:

```
CREATE TABLE f(
   id INT PRIMARY KEY     NOT NULL,
   i  TEXT                NOT NULL,
   j  INT                 NOT NULL,
   k  CHAR(50),
   m  REAL );
```

will generate the following tree which as shown in figure 6.5. Although the tree has been displayed in landscape format, it was still too large to be displayed fully, but shows the table name, and column definition part of the parsed SQL structure in tree form.

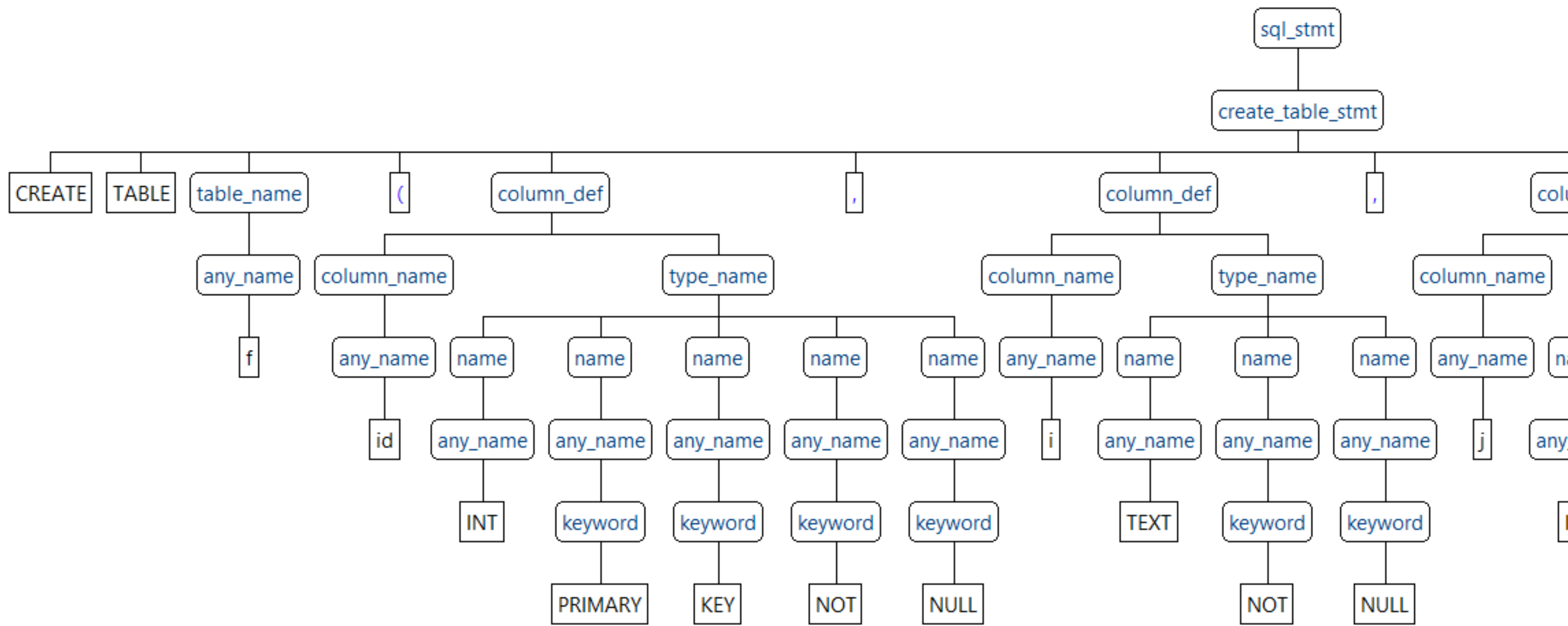

Figure 6.5: SQL Tree - Create Table

6.4.4 Parsing Uniface Grammar

Uniface grammar can be extracted from available documentation and subject matter experts. However, it is challenging to formalise its grammar rules. This is because 4GL atomic operations are encapsulating procedures and thus producing ambiguity and several interpretations. Figure 6.6 below shows an example for extracting and understanding *If-block* Uniface syntax.

The example given above is taken from an existing statement for creating a table, as provided in the exported XML-like code originating from the ESIS Uniface system that this research has been largely based on. However, the table and some field names have been omitted due to data-privacy sensitivities. The results achieved when parsing SQL have been based on its standard as defined in the ISO-IECS Standard 9075 [259]. This technique was successful and paved the way for extending the technique directly on Uniface syntax.

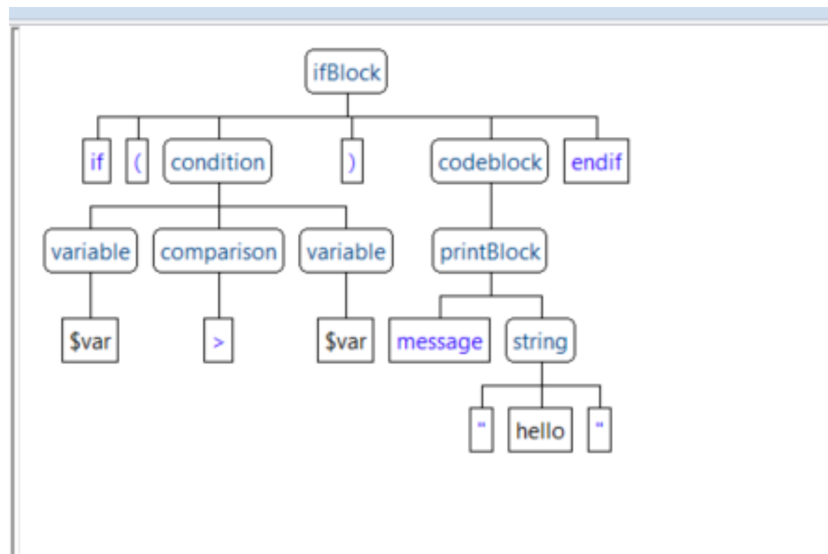

Figure 6.6: Uniface If-Block Parse Tree

This *If-block* showing in figure 6.6 contains a code block. The coding block is built within one operation that displays a line of interpolated string.

This code-block is a recursive enumerable abstract structure that represents graphically the code syntax. It is used to visualise the language syntax and recursively translate each node in the tree from the source language to the target language.

6.5 Conclusion

An unsuccessful attempt was made to export the XML code into SQL (section 6.2), which foundered on SQL row limit restrictions.

This chapter has covered successful techniques used to extract Uniface elements.

The previous chapter, (chapter 5), had shown how the data structures (schema) of Uniface XML-like code could be extracted and visualised as either DTD scheme or as XSD schema, the latter containing a richer set of information.

This chapter builds on that by extracting elements from the valid schema, and makes a further refined XML file which can then be properly formalised and parsed using Grammar Formalism (as described in chapter 7).

Chapter 7

Implementation IV - Grammar Formalism

With a data structure determined into schema (chapter 5), and the XML-like code having its Uniface elements extracted (chapter 6), the XML code is now in a position for formal grammar rules to be applied to it.

This will enable the extraction of the underlying structure of the source system (in this case the University of Essex ESIS system) in order to convert (re-engineer/reverse engineer) it into another language (in this case C#).

7.1 Principles of Grammar Formalism

This section explains the formal methods and principles that researchers use to construct formal grammar. In addition, it explains how this research applied relevant methods for Uniface grammar.

7.1.1 Domain Specific Languages (DSL) Need Regular Expressions + Extensibility

Domain specific languages (DSLs) started and became popular with Unix development, see Cunningham [260] and Zdun [261], but have not become as popular as object-oriented languages. DSLs began as a collection of packages that assisted developers in implementing programs more rapidly as set out by Fowler [36]. In most cases, DSL knowledge was withheld by the vendor(s) and owner(s). Since around 2000, DSLs have been more popular with developers because of their speedy implementation, stability, and widespread educational resources, Damyanov and Sukalinska [262].

Object-oriented languages have been more widely used having improved on compiler traits and optimisation techniques.

DSLs can work as a wrapper for libraries or frameworks. Even a skilled current generation programmer, when attempting to use a DSL, might struggle because the underlying 4GL/DSL concepts are unfamiliar to those brought up on object-oriented programming languages and concepts.

Alternatively, developers may decide to separate configuration from implementation in their system to enable dynamic behaviour at run-time. To do so, it is very common to use XML, a language built for extensibility, which eases the creation of a custom vocabulary. This has led to increasing complexity of configuring a system. DSLs had been specifically designed to overcome similar hurdles.

Designing simple DSLs using regular expressions may suffice, according to Dubé and Feeley [263], and Parr and Quong [83]. Regular expressions are also used to build DSL tools, (Cook et al. [264]) to find occurrences of a particular pattern of characters.

As an example of regular expressions, an assertion can be made that the following expression statement is required: ***that matches a 'd' pattern if, and only if, the char 'd' is not followed by 'r'***. This can be expressed in formal notation as:

$$RegularExpression = d(?!r)$$

Amending this regular expression can add another level of complexity to code maintenance. However, wrapping expressions with a regular expression control engine would eliminate the challenge. Regular expression tools have significantly gained in popularity and this has created the impetus to enhance the search pop-ups used in text editors by allowing users to switch modes from normal text search to regular expression search mode, Hosaya and Pierce [265].

The next section discusses the benefits of using modular grammar with DSLs for reusability.

7.1.2 Modular Grammar

7.1.2.1 Applying Software Engineering (Modularity) to Grammar Engineering

In software engineering, modular programming has been defined by the IEEE as:

> modular programming. A software development technique in which software is developed as a collection of modules (IEEE Standard Glossary of Software Engineering Terminology, Radatz et al. [266]).

and modularity is defined as

> The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components, Radatz

et al. [266].

Some work has been conducted by Alves and Visser, 2008 [267] on applying software engineering methods to grammar engineering, to overcome the known issues with traditional parser builders.

Language-oriented programming, Ward [268], promotes a method of writing domain specific languages which abstract the knowledge and known algorithms of the domain and host it within the language semantics, Şutîi et al. [269].

7.1.2.2 Language Workbenches

The development of language workbenches has empowered developers with integrated development environments (IDE) tools which can assist in the process of writing DSLs, as described by Fowler [270].

Language workbenches (Fowler, [270]) provide the necessary tools such as meta-language tools to edit, generate code, add constraints, and data types definition. Nonetheless, in order to convey the vision of language oriented programming, meta-language workbenches have to offer integrated features of modularity and reusability. For example, having a reusable arithmetic expression engine would reduce the development time for creating new DSLs. It would enhance the stability and quality of a DSL's function library by allowing more iterations of tests to run in a given time. Modules need to be designed so that they are granular and respect integrity, Johnstone et al. [271].

Language oriented programming and grammar composition research enables researchers to find potential solutions to enhance the usage of DSLs in the form of dependent libraries that can be embedded in other languages. For example, research in, and developments of, embedding SQL in object-oriented programming languages, De Lucia et al. [272]. Additionally, such an approach makes it possible to extend the DSL language semantics without amending the core implementation.

7.1.2.3 Attribute Grammar Specification Language

Attribute Grammar Specification Language (Paaki [273], Van Wyk et al., 2007 [274], like a DSL, has the advantage of providing general purpose features. 'Silver', an extensible attribute grammar specification system, Van Wyk et al., 2010, [275] implements pattern matching, collection attributes, and constructs which fall under the category of DSL general purpose application features. Silver uses a LALR parser generator, named 'Copper'.

Developing such tools can help non-experts to provide extensions to languages. Using deterministic and context-free grammar, Schwerdfeger and Van Wyk [276] created a modified version of a LR parser and a context aware scanner. The authors assert that grammar conflicts are detected when the parser is generated. Detection by the parser is a late stage in the processing for

conflicts to be detected. Copper enforces the generation of LALR grammar and therefore it rejects non-composable grammar with reasons.

7.1.2.4 Reference Attribute Grammar

Using Reference Attribute Grammar (Knuth [277]) and object orientation, ‘JastAdd’ Ekman and Hedin [278], Hedin [279] implement tools to generate languages and parsers. JastAdd allows the generated tools to be extensible. An example of such a tool that JastAdd implements, ‘JastAddJ’, is a fully functional Java and extensible Java compiler. While traversing the generated parsing tree, the tools allow semantic actions to be performed. The AST is extended using static aspects.

ALL(*) is used by ANTLR which uses a depth first import process. ANTLR claims to overcome the issues associated with adopting the Generalised LR (GLR) parsing algorithm and Parser Expression Grammar (PEG) by using the approach of context-sensitive languages Parr and Fisher [130].

7.1.2.5 Parser Expression Grammar (PEG)

PEG and context-free regular expressions (REs) come with ambiguity when determining a grammar execution path. Therefore, PEG (Ford [254]) provides an alternative solution. It eliminates the ambiguity by selecting the first matching role and offers unification to both lexical and hierarchical syntax in the defined grammar.

Grimm [280] utilised the benefits of modularity and introduced the tool ‘Rats!’ which uses PEG to generate parsers. Rats! uses a grammar which is more expressive than PEG (Ford [254]). In addition, it promotes grammar modularisation and allows extensibility. The Rats! generated parser uses variables such as $yyBase, yyResult$ to scan the code and generate results. When post-parsing a rule. The semantics of an object become the semantics of the production rule. Rats! grammar expressiveness shares syntactical traits with C++ -like syntax.

7.1.2.6 Parsing a Byte String - Example Code

Parsing a byte string in Rats! can be modularised as the following example shows:

```
module ByteString;

body {
  Result parseChars(String number, int start, int base) throws IOException {
    int n;

    try {
      n = Integer.parseInt(number);
    } catch (NumberFormatException x) {
```

```
      return new ParseError("Malformed length", start);
    }

    StringBuilder buf = new StringBuilder(n);
    for (int i=0; i<n; i++) {
      int c = character(base + i);

      if (c != -1) {
        buf.append((char)c);
      } else {
        return new ParseError("Unexpected end of bytestring", base + i);
      }
    }

    return new SemanticValue(buf.toString(), base + n);
  }
}
```

7.1.2.7 Other Parsing Examples

The Rats! parser generator was used to parse the Wikipedia Object Model Dohrn and Riechle, 2011a, [281], Dohrn and Riechle, 2011b [282] and is also used to generate a fast parser, Kuramitsu, 2016a [283] along with Nez grammar, Kuramitsu, 2016b [284].

7.1.2.8 Using Algebraic Transformations

Algebra transformations adhere to catamorphism concepts (where there is just one algebraic structure for a given input), Németh [285]. Therefore, algebra can extend languages, (Andersen and Brabrand, 2010 [286]) using catamorphism and context-free grammar transformation. Defining languages using syntactical algebra exploits the potential of extending languages in a modular fashion. Such tools have been implemented by Brabrand and Schwartzbach, 2007 [287], Brabrand and Schwartzbach, 2002 [288], Andersen et al., 2013 [289], Andersen and Brabrand, 2010 [286].

These approaches highlight the benefit of techniques embodying algebra to extend languages as the language construction process is constant and therefore it can be handled at compile time.

7.1.3 Grammar Engineering for Uniface

The research in this thesis follows a tool-based approach in which the generated language is made up of grammar-centred development using language-kit tools and empowered with parser generators.

For this study, the Gold Parsing System [290], Cook [291] was chosen together with Backus–Naur Form (BNF) or Enhanced Backus–Naur Form (EBNF) Wirth [256] to write the grammar syntax.

Further detail have been provided for Uniface's grammar rules as coded by the author, in both Gold Parser (section 8.3.2) and in ANTLR (section 8.4).

For this research the author designed and built the tool in three phases (the ANTLR implementation has been referenced here):

- Phase one - creating Uniface grammar, (section 8.4.4),
- Phase two - generating the parser, (sections 8.4.5, and 8.4.6),
- Phase three - building the visualiser tool, (sections 8.4.8 and 8.10).

Another approach to Phase one would have been to use Syntax Definition Formalism (SDF) Heering et al. [292], Lämmel and Wachsmuth [293] or extensible meta-programming languages such as 'SugarJ' or 'Racket', both of which are covered by Zaytsev [294].

7.1.4 Grammar Correctness

Grammar is built on phrases and after each phrase is processed, there needs to be a stage of refactoring and fixing meta-syntax.

In the absence of any formal definition of Uniface statements and keywords, semantics for this thesis were constructed from the system behaviour. Therefore, the generated system and source system should semantically exhibit similar behaviour.

However, measuring the system's grammar for accuracy can be a challenge in itself and has not been attempted in this research. However, since the system output matches the intended behaviour, as far as can be ascertained, it is believed that the author's grammar syntax is correct.

In each phase, listed in section 7.1.3 above, test units were included to test the written grammar using black-box and white-box methods.

The Gold Parser, used in this research, provides a tool to test the input code against the grammar and generate an AST on the fly and so this tool was utilised in this research to generate initial test results for the grammar.

7.1.4.1 Grammar Engineering

Software engineering techniques can assist in building and developing grammar. This means that writing grammar can follow some of the software life cycle including:

- Version control,
- Static analysis, and

- Testing.

Grammar syntax is an important phase for the parsers to generate an optimal parsing tree. Therefore, creating language development tools is directly related to the quality of the grammar it is based upon Alves and Visser, 2006 [295], De Jonge and Visser [296], and Alves and Visser, 2008 [267].

Grammar engineering, Alves and Visser, 2006 [295], is the field where software engineering methods can be applied to grammar generation. Traditionally, grammar has been built into parser generators for a specific language, meaning that the generated parser is built for the language semantics of the written language. For example, the following tools, 'Yacc', 'ANTLR', and 'JavaCC' use LALR or LL(1), and are tightly coupled to the input grammar.

This enables the generated parser, which constructs a compiler, to:

- Compile programming code,
- Generate AST trees,
- Provide Metrics computation,
- Serialize/deserialize,
- Traversal support and
- Pretty printing.

Some of the techniques are also can have difficulties, according to Van Den Brand et al. [297]. The mainstream techniques are not designed to create conflict-free grammar and do not support a long-term maintainability process. Special purpose programming languages, for example TXL, are designed predominantly to service computer program analysis and transform source code. This language implements advanced pattern matching to deliver its function. TXL enjoys the benefit of BNF grammar and functional programming to deliver the description of a targeted structure and set of applied rules. TXL is used to analyse embedded SQL queries within the PHP programming language. This combination is widely used to build dynamic websites, retrieve the content from relational databases using SQL, and model them, Anderson [298]. Text analysis and pattern matching can also be used for code refactoring and metrics and discovering security vulnerabilities.

7.1.4.2 Rascal

Rascal (Vinju et al. [299]) has been developed to overcome the shortcomings of the other tools in the same field. It can be used to analyse software as well as to transform the source.

The basic technique used in Rascal is Syntax Definition Formalism (SDF). It also uses Algebraic Specification Formalism (ASF), as described in Bergstra et al. [300], which formalises the definition of abstract data types.

Rascal has been designed with the following features:

- Rewriting rules,
- Closures,
- Higher-order functions,
- Comprehension,
- Generators,
- Generic traversal,
- Pattern matching,
- Imperative core with immutable data.

7.1.5 Algebraic Specification Formalism (ASF)

ASF is modular and modules adhere to the rules of 'Using Component-Based Methodologies Towards Reverse Engineering Uniface' as explained in section 3.3 and S.O.L.I.D. principles as explained in section 8.

Their implementation is demonstrated as shown in the example below:

```
module M1
  begin
    exports
    begin sorts A, B
      functions a: -> A
      f: B -> B
    end
  end M1
module M2
  begin
    parameters
    P begin sorts C
      functions c: -> C
    end P,
  Q begin sorts D
    functions g: D -> D
  end Q
end M2
```

Source transformation has also been used to detect software clones, Lämmel [301]. Code refactoring

tools have been also implemented using source transformation and concrete syntax trees (parsing trees) Bravenboer and Visser [302] in order to manipulate code using code templates.

Grammar is built in iterations. Each iteration may need refactoring and fixing meta-syntax. In the absence of any formal definition of Uniface statements and keywords, semantics have had to be constructed from the system's behaviour. Therefore, the generated system and source system should semantically exhibit similar behaviour.

7.1.6 Syntax Definition Formalism

SDF is based on context-free grammar and it extends BNF to be more like a pattern expression style than EBNF. SDF enables syntactical description of existing programming languages, such as C++ or Java, and also enables the blending of different programming language's grammar or extending the current ones, Erdweg and Rieger [303].

However, SDF is not limited to programming languages but can also be used to provide formal definition for domain specific languages, thus offer the semantic analysis of programming languages to DSLs. SDF supports modular grammar specification, which makes it possible to parse embedded languages and language dialects for the ones that share a common syntax base, Johnston et al. [271]. This means that writing grammar can be freed from LALR(1) or LL(1) grammars. Unlike context-free parsers, SDF disconnects the meta-language definition from the parser-specific implementation of the defined language using lexical and syntactical grammar.

In addition, SDF provides a mechanism to write mathematical disambiguation rules to resolve ambiguous grammar. This is done using an SLR(1) parse table generator and a Scannerless Generalised LR parser (SGLR), the steps for which have been shown in figure 7.2 below.

7.1.7 Parsing Ambiguous Grammar - An Example

An example of ambiguous grammar is given below:

```
<exp> ::= <exp> "+" <exp>
       |  <exp> "*" <exp>
       |  "(" <exp> ")"
       |  "a" | "b" | "c"
```

This grammar is syntactically valid in languages like C, however, the parser can have two distinct paths to follow in order to execute these expressions. This example of such ambiguous grammar expression paths has also been shown in the form of a tree in figure 7.1.

SDF identifies ambiguous possibilities in order to make explicit decisions only. SDF generates the parsing tree and optionally maps to an AST. SDF implements:

- A parse-table generator, and

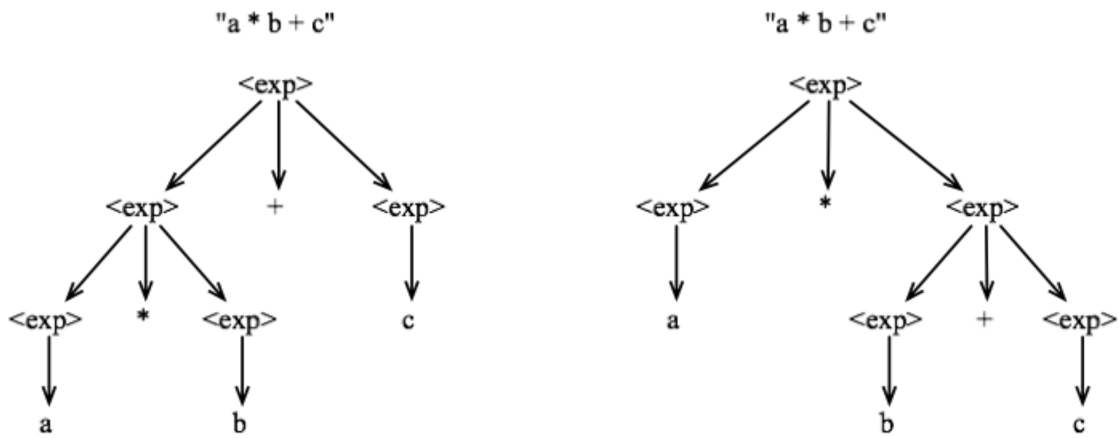

Figure 7.1: Ambiguous Grammar: Expression Paths

- SGLR: a scannerless generalised LR parser (as shown in figure 7.2).

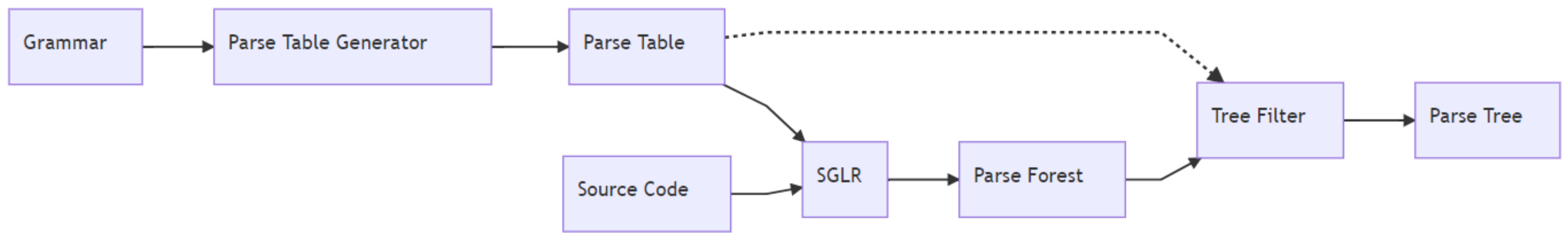

Figure 7.2: Steps for Generating Scannerless Generalised LR Parser (SGLR) and Parse Tree

7.1.8 Template Meta-Programming and Templates

Template Meta-programming (TMP) is code that instructs the compiler for a specific task (e.g. to create an AST). The template may contain variables of a generic data type. The type can only be determined at compile time which then cast the generic types into a specific ones, which enables code generation according to the programming language grammar rules and data-types.

Template meta-programming is classified under meta-programming, Sheard and Jones [304]. Template meta-programming is a technique that allows stringified source code to be passed into the code as parameters and arguments. This code is then used to generate program segments. Template programming is supported by several languages such as C++, Abrahams and Gurtovoy [305] and the D language.

String templates are used for generating text from pre-defined templates. It can improve encapsulation and improve the separation between model and view. It also enhances security and maintenance. When working on string templates, researchers and developers can improve single point of change solutions. For example Gutkin and King [306] have used the concept to generate inductive reasoning for spoken languages. Code generation is a major area where the concept of string templates are used, Syriani et al. [307], Bazelli and Stroulia [308].

String templates are also used for structured generation of code. This makes it immutable and restrictive when used with 4GLs. Defining accurate grammar for a 4GL can enable provision of structuring templates that deliver domain specific technologies.

Lämmel [309] has considered this method for prettifying code and managing repositories. Automatic tools such as interpreters, translators, analysers, transformers, and pretty printers could create n new instances to perform the desired tasks based on timed schedulers.

According to Shrestha [310], a process oriented design application such as 'ProcessJ' has depended mainly on string templates to generate source code, web pages and formatted text.

Figure 7.3 shows the parsed If-block of Uniface code. In this block the types 'variable', 'comparison' and 'printBlock' may be defined as templates to allow the compiler to reuse these templates when it encounters the next occurrence of a similar code block. The corresponding function is shown in appendix I.

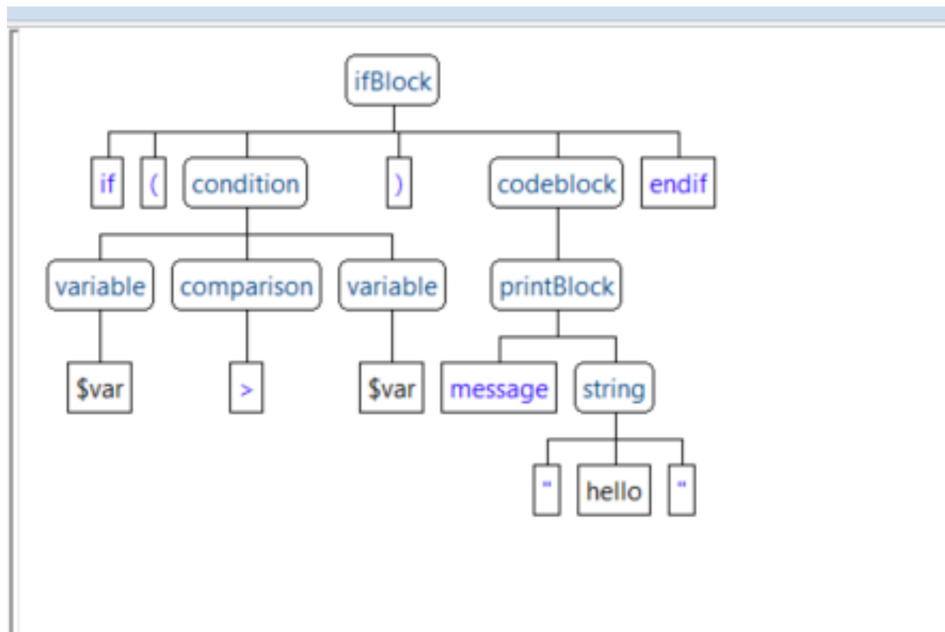

Figure 7.3: Uniface If-Block

This thesis's research has been focused on reverse engineering 4GLs; it aims to utilise string templates for the purpose of generating any of the following:

- AST for parsed syntax,
- Data Model,
- Documentation,
- Generated code for the targeted language,
- Structured input for the knowledge base.

7.1.9 Using BNF/EBNF for Syntax Notation

BNF which stands for Backus–Naur form, Wirth [256], was derived from Wirth's syntax notation. This research uses EBNF which extended Backus–Naur form to deliver context-free grammar notation.

Originally Wirth's syntax was used to express data-modelling for STEP (International Standard for the Exchange of Product Data), which was used for the representation and exchange of product manufacturing information, Zhou and Naghi [311], Hardwick et al. [312].

Formal languages are frequently represented by EBNF as a sequence of symbols which match the language rules, Wirth [256].

The novelty of this research is that it has reverse engineered a proprietary 4GL by creating the formal representation of its syntax using EBNF grammar rules. The same applies to the representation of the syntax of a proprietary 4GL in the form of syntax diagrams or "railroad diagrams" as explained in section 8.4.4.

7.1.10 Syntax Diagrams

Syntax diagrams are a pictorial but formal representation for context-free grammar, Braz [313], which are suggested as an alternative for BNF or EBNF.

Parsers, such as that built here (sections 8.3.2, and 8.4.4), which use LL(k) and context-free grammar, work best with syntax diagrams to generate the parsing table from lexicals and tokens (section 3.4.1). Due to the formal nature of these diagrams, only one interpretation of the input can be valid.

7.1.11 Overview of System as Implemented

This section aims to illustrate the process of translating the behaviour of the system and understanding the logic embedded in the targeted system source code.

Template programming has also been used to develop embedded domain specific languages, e.g Embedded Domain Specific Language (EDSL), Shioda et al. [314] and other different types of applications, Hollman et al. [315], Curtin et al. [316], Reinert et al. [317]. Kourounis et al. [318] have shown that template programming could be used in the domain of differentiable/differential functional programming for the compile time symbolic differentiation of algebraic expressions.

This research provided an implementation of ANTLR used with Template Meta-Programming in Appendix I

System architecture and design conformance was described in the discussion on static modelling in section 4.2.2.

In order to demonstrate a complete translation of the parsed source code of the 4GL from end to end, the same example is used as before 6.4.4.

The system as built by the author for this thesis, has been divided into three main software logical layers where each layer corresponds to the set of functionality illustrated in figure 7.4.

- Front-end logic as an object-oriented system,
- Middleware or connectivity layer,
- Back-end or data source layer.

Figure 7.5, depicts the topological architecture of the system as developed.

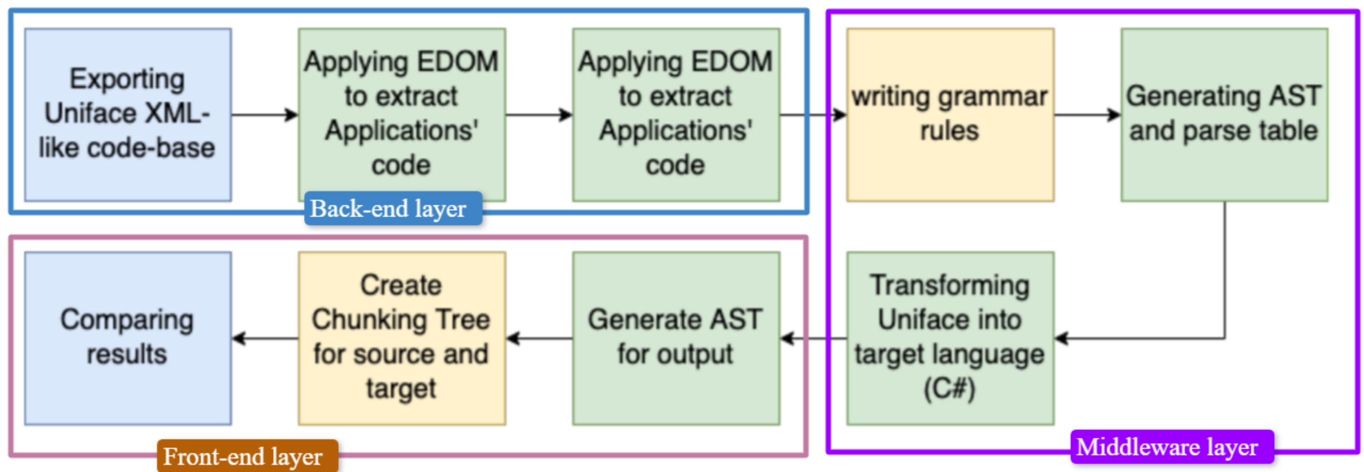

Figure 7.4: General Overview of the System's Components Grouped by the Relevant Layer

Figure 7.5: Topological Architecture of the System

The system has been designed in this way so that diverse data could be accessed in a common way to a standard format, instructed by the meta-data provided by the middleware layer.

Based on this n-tier architecture, the service is responsible for reading the entities and entity sets, where an entity set is a list of entity records. Figure 7.6 shows how the class diagram is auto-generated, based on the entity relationship definition in the middleware layer and illustrates the Object Relational Mapping (ORM) set out in section 2.4.

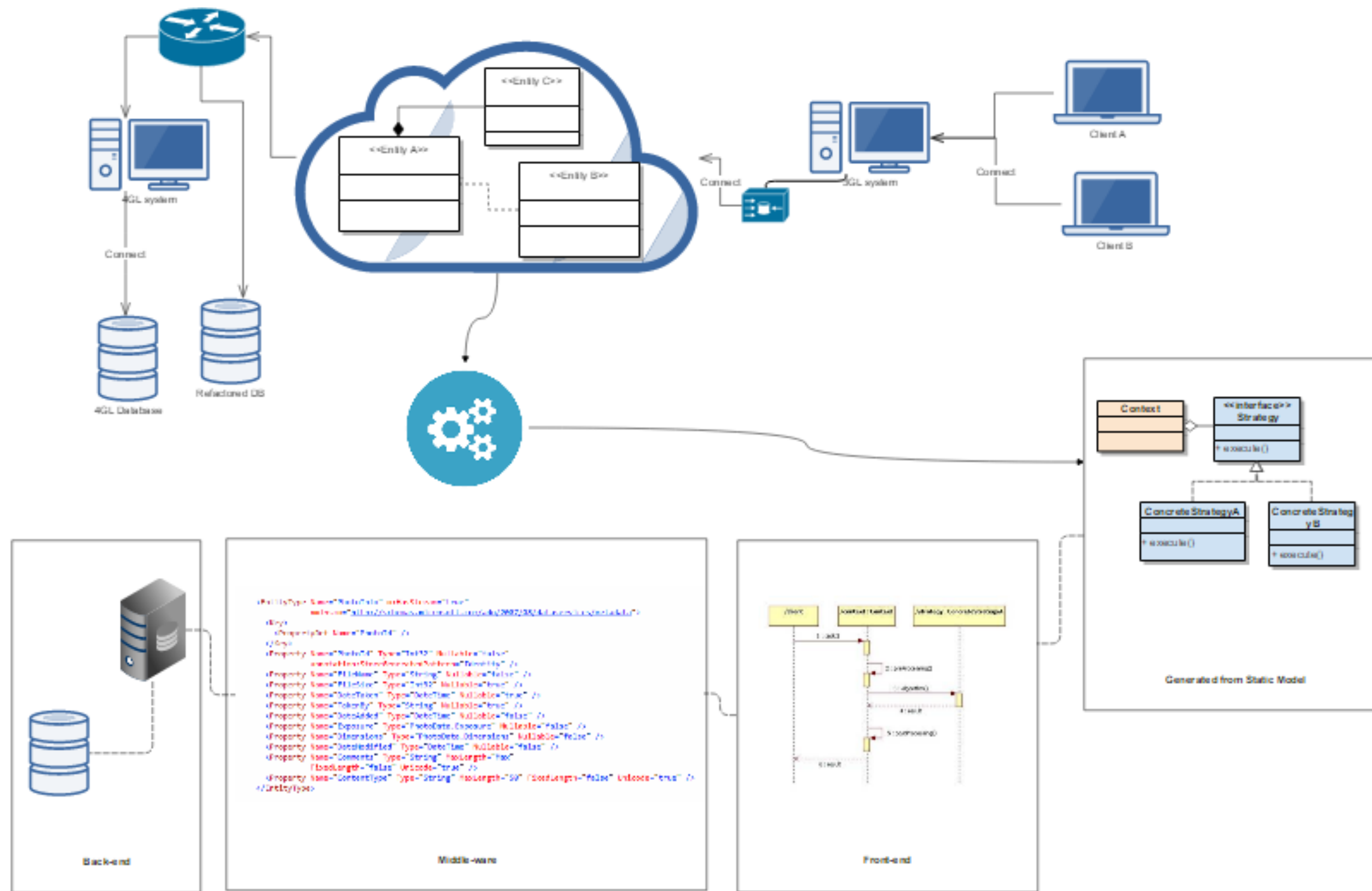

Figure 7.6: Object Relational Mapping for the Middleware Layer

Object-oriented methods enable the extraction of targeted system artefacts and the building of new object-oriented models. Object-Oriented Analysis and Design (OOAD) is the branch of software engineering that studies the practise of extracting system designs in an object-oriented manner to aid the process of development, Rumbaugh [319]. Modelling the system in an object-oriented manner requires a good understanding of the mapping problem (what items mirror or map to other items) and semantics. Modelling can be achieved independently from software visualisation such as a class diagram.

However, in order to model the system, the following components are mandatory:

- Class,
- Entities,
- Operation,
- Value and
- State.

In addition, the relationships between the static and semantic components needs to be established.

7.2 Data Flow from Front-end to Back-end Layer

The back-end database (meta-data database) needs to be designed to store the data which has been extracted from the legacy system data source. The fields in the meta-data database should have a reference to the records from which they were copied and a data type which matches the data type of the same record in the original legacy system (in this case the ESIS system). The data could have been scattered over different data sources in the legacy system. The meta-data database has to track all Create, Read, Update, Delete, and Query (CRUDQ) operations on the legacy database. CRUDQ operations are triggered by the front-end system or other external systems which may need access the internal database such as the finance system.

Continuous engineering needs to be carried out during migration from the legacy system to the new one. If the data is to be replicated in the new system, there will be a danger that the data and structures of the two databases, that is the input source and the destination, may get out of step with each other. It is best, therefore, to first build a meta-data database that holds information about the legacy system data store and connection strings.

The connection to a database or data store is specified by the connection string which contains the information needed to connect to a data-source. This information includes the server-name, driver meta-data, and could also include security information such as the user-name and password.

The meta-data database can then be used in automated form, to access the content, update it, add to it, or truncate it.

In order to achieve this, an engine was built to process the requests between the front-end system and back-end via the service layer.

The back-end system is blind to the front end data structure and relationships between the data. However, the engine is language-specific and relies on the selected framework (C# in this thesis) to manage user requests, pull data, and fetch it back.

User requests are passed from the front-end using a unique identifier (the service reference) with an associated action code appended to it (e.g. whether a Create, Read, Update, Delete or Query [CRUDQ]).

The service controller is the entry point for requests. It validates the request and its arguments. The business logic can either be handled by the service controller or deferred to the back-end layer and listened for a response.

The service controller is responsible for interpreting the destination entity or the back-end router (as shown in figure 7.6), should redirect the request to the run-time code of the back-end. Table 7.1 lists the standards followed to establish the connections across layers and exchange data.

Reference	**Description**
[RFC 2616]	The HTTP protocol
[RFC 5023]	The Atom Publishing Protocol
[OData: Core]	The core specification for OData
[OData: Atom]	Atom format conventions for OData services
[OData: JSON]	JSON format conventions for OData services
[OData: URI]	URI conventions for OData services

Table 7.1: OData Standards

7.3 Conclusion

Chapter 7 has taken the process of reverse engineering the Uniface system, into a lower level, formally checking the grammar of the incoming XML, to avoid ambiguities, using 'Syntax Definition Formalism' and BNF notation.

It also has discussed the crucial division of the proof of concept system into Front-end (object-oriented system), middleware (connectivity layer) and back-end (data source) layers. This enabling the end user using the front end software layer avoiding having to deal with the intricacies of the back-end which is the data source.

The chapter that follows, (chapter 8), discusses the overall design principles used, the high level language framework used, together with detail about the two main parser tools (Gold Parser and ANTLR).

Chapter 8

Using Parser Generators to Implement Uniface Grammar

This chapter discusses the underlying design principles for the implementation of this research (section 8.1 - S.O.L.I.D. Principles).

It then covers the language frame work used (section 8.2 - .NET Framework and C# Programming Language) before moving on to discuss in detail implementation and coding using two different parser tools:

- Implementation in Gold Parser (section 8.3) and
- Implementation in ANTLR (section 8.4)

together with a comparison between the two parser tools (section 8.5).

Lastly in this chapter, section 8.6 provided a comparison of some parsing strategies as measured by complexity measures, for the principal parsers (Gold Parser and ANTLR) used in this research.

8.1 S.O.L.I.D. Principles

This research contains modules that were implemented using design principles drawn from:

- Component-based Software Engineering (section 3.3.2) and Component-based Design (section 3.3.3),
- Design patterns (See sections 2.3.1 and 8.4.11) and
- S.O.L.I.D principles (also known as Agile principles), Martin, R. and Martin, M. [320] - as discussed immediately below.

The S.O.L I.D. Principles, according to Martin, R. and Martin, M. [320] are

- Single Responsibility Principle (each class in the design deliver a sole purpose and should be extended for the purpose it serves),
- Open/Closed Principle (a class in the design can be extended but not modified where possible),
- Liskov Substitution Principle (classes can inherit from the base class. However, child classes must be written so the base class can be replaced with any descendant),
- Interface Segregation Principle (each class should implement an interface that prototype the purpose it serves),
- Dependency Inversion Principle (classes can depend on abstract class but not concrete implementations).

8.2 .NET Framework and C# Programming Language

Microsoft 'dotnet' or '.net' is an open-source platform established by Microsoft in the late 1990s[321]. The framework enables developers to build applications for the web, mobile, desktop, gaming, and the Internet of Things (IoT). There are several programming languages which are supported by '.net' such as C-Sharp (C#), F-Sharp (F#), and Visual Basic (VB).

C# was selected for this research and used as the principal programming language for the code for all three levels of the author's proof of concept system.

C# is an object-oriented programming language and type-safe language. In addition to the native resources that the '.net' platform offers, other packages are available, such as the 'NuGet' package manager. This contains a list of dependency packages which developers can access for reusability. For this research, the entity framework 'ado.net' , Adya et al. [126] was downloaded and used via 'NuGet' manager.

C# (Wiltamuth and Hejlsberg [322]) supports the Component-Based Software Engineering (CBSE) methodology, which was explained in section 3.3.2.

Declarative programming, which structures an application without defining the control-flow, can be composed with methods, attributes, and properties. The language provides a powerful garbage collector that destroys unused objects. Type-safe features enforce initialisation of variables and valid casting by the developer. Exception handling is extensible for catching errors of specific types.

The hierarchy of an application consists of the following concepts:

- 'programs',

- ‘namespaces’,
- ‘types’,
- ‘members’, and
- ‘assemblies’.

Assemblies embed types and members such as classes and fields. When the C# compiler packages an assembly, it produces a binary file by parsing and compiling namespaces contained in the assembly. The executable code is formed in Intermediate Language (IL) format. The just-in-time compiler (JIT) translates IL into a processor specific executable (.EXE) program.

The C# code output in appendix C (C# Code to Transpile Uniface into C#), derives from C#'s capability of supporting multi-file structures. This is where the code has been distributed over several files to present a logical separation of components.

The C# language supports the following primitive types:-

- ‘bool’, ‘
- ‘byte’,
- ‘char’,
- ‘decimal’,
- ‘float’,
- ‘object’,
- ‘sbyte’,
- ‘string’.

The code in appendix C implemented a Transpiler which here translates types automatically from the Uniface language into C# is feasible syntactically, however, the semantics between the two languages can differ.

8.3 Implementation in Gold Parser

This section shows how Uniface grammar rules were implemented and debugged. It also shows how the implementation results were output in Gold Parser.

8.3.1 Parser Operation

Parsers use sequence of characters as input to generate an abstract syntax tree (the concept was discussed in section 3.4.3). This sequence of characters is fragmented into tokens. Tokens are classified as terminals (node-leaf in the syntax tree) or non-terminals (parent nodes in the syntax

tree). An example of a syntax tree was given in figure 6.6. LALR parsers are designed to statically analyse a valid BNF grammar and identify sub-strings which are convertible to non-terminals. This static analysis boosts the parser performance and reduces the number of errors at run-time.

8.3.2 Gold Parser

Gold Parser (as previously mentioned in sections 3.4.6.3 and 7.1.3) is a tool that uses LALR parsing and provides either a user interface or input at run-time to write BNF grammar, generate the parser and test the output. It can be extended with 'Engines' components that generate a wrapper in multiple programming languages to access the parser by means of other programming languages.

This section illustrates an example of parsing a Uniface syntax statement in Gold Parser.

Writing grammar rules and syntax requires human expertise, and therefore is a semi-automated process in the author's system.

The 'Builder' is a component built into Gold Parser which constructs the parsing table from input grammar rules in BNF. The 'Builder' outputs an Enhanced Grammar Tables file for the 'Engine' to act on.

The algorithm uses a deterministic finite automaton (DFA) approach to determine the next valid step for the parser. Regular languages, for example, use regular expressions to draw a DFA tree for a given input. There is a single path to take through a DFA tree, for valid input to lead to its successful execution.

A DFA is an ambiguity-free model. This means that the transition from any given state against the input is deterministic, that is, has a single interpretation. The number of states that a DFA can have must be finite. Moreover, the number of transitions over states for any given input must also be finite. A DFA algorithm uses the input to determine the next step for a transition.

A DFA can be represented using a graph. A graph is an abstract data type that is constituted from a set of vertices and a set of edges, Godsil and Royle [323]. Gold Parser incorporates a DFA to implement the tokeniser which validates a series of characters to assert that a string qualifies for token rules.

For a sample Uniface snippet such as:

```
1  sort/e "students.entity"
```

a DFA can be generated such as in figure 8.1:

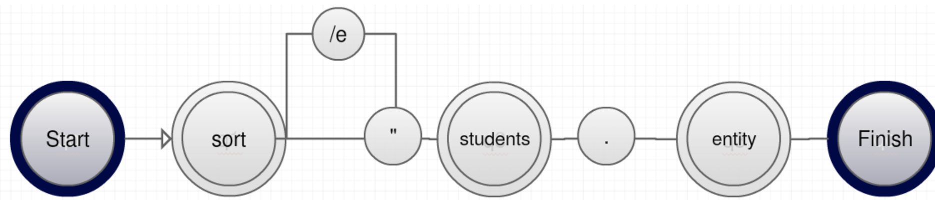

Figure 8.1: Gold Parser - Deterministic Finite Automaton for a Sample Input

Figure 8.2 displays the log generated for the successful examination of the available routes based on the generated DFA:

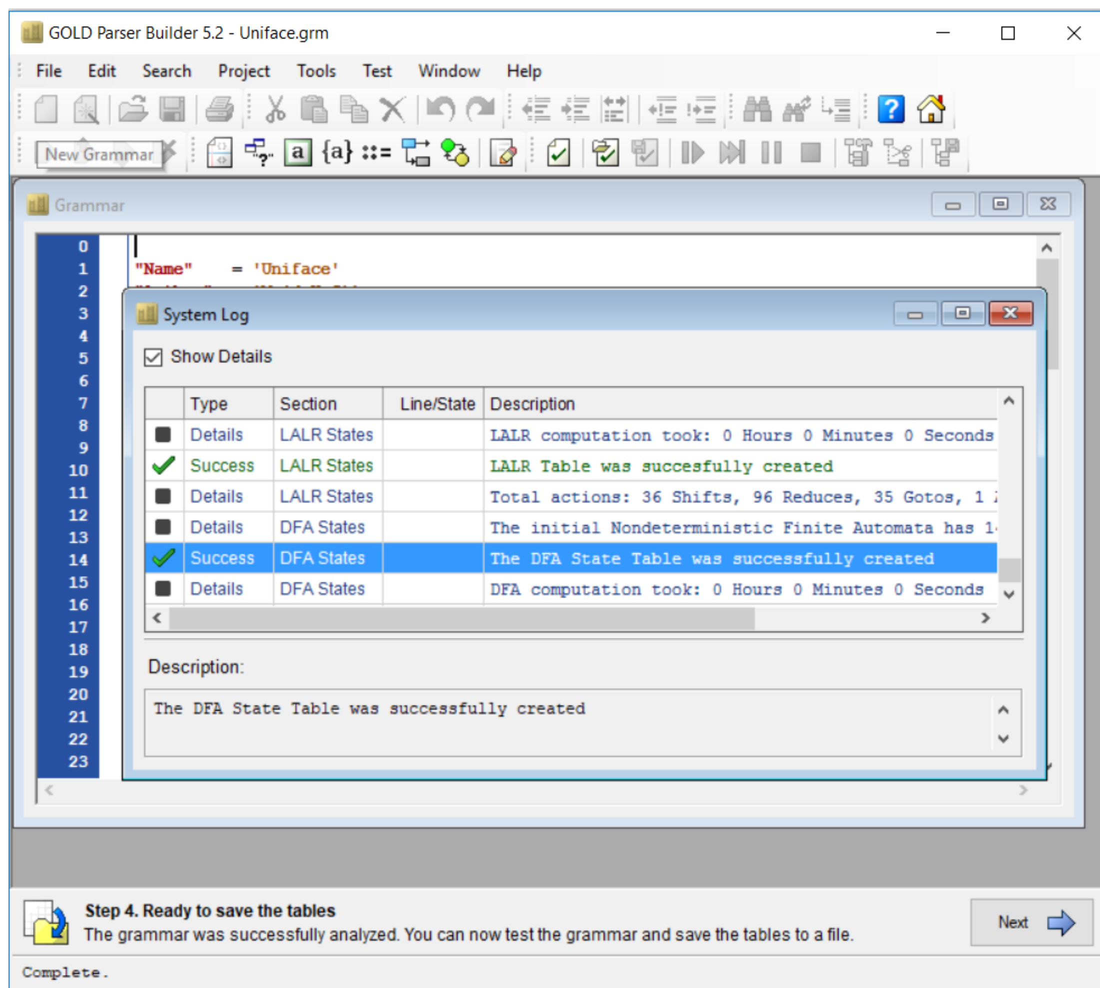

Figure 8.2: Gold Parser - Deterministic Finite Automaton Screen

8.3.3 Functions and Parameters

Some selected examples of Gold Parser functions and parameters are described as they are used further below, in an example of using the Gold Parser in practice.

The function 'sort' shown in figure 8.1 can be accessed using onscreen input or by providing '/e' as an optional parameter when calling the Gold Parser using a programming language.

The parameter "*students.entity*" is a mandatory input of data-type 'string'. The string literal starts and ends with double quotes '"'. The dot '.' in the string separates the table name 'students' from the entity name, in this case 'entity'.

8.3.4 Tokeniser

The tokeniser is designed to match a series of characters which must match a pre-defined grammar rule. In the given example in figure 8.1 the keyword 'sort' is matched after the tokeniser scans the individual characters 's' 'o' 'r' 't'. When the full keyword 'sort' is matched, the tokeniser creates a token which groups these characters into a single token 'sort'.

8.3.5 Multi-language Support

Gold Parser's support of multiple programming languages is achieved by decoupling the parsing table construction logic from the algorithm that performs parsing. Gold Parser tables are constructed with DFA and LALR parsing techniques. 'Engines' are implemented in Visual Basic, C++, C#, ANSI C, Java and Delphi. These 'Engines' are made available on the official website for developers to use. For this research the language C# was selected for implementing a Uniface compiler with Gold Parser and the 'parser table' was generated accordingly.

8.3.6 Table Output as .egt File

Gold writes parser tables into a special file type named Enhanced Grammar Tables file that have the extension '.egt'. The file is agnostic to both programming languages and also to operating systems. This enables engines of any programming language to interact with the content of the file. Gold Parser features a save dialogue screen for saving the '.egt' file into permanent storage.

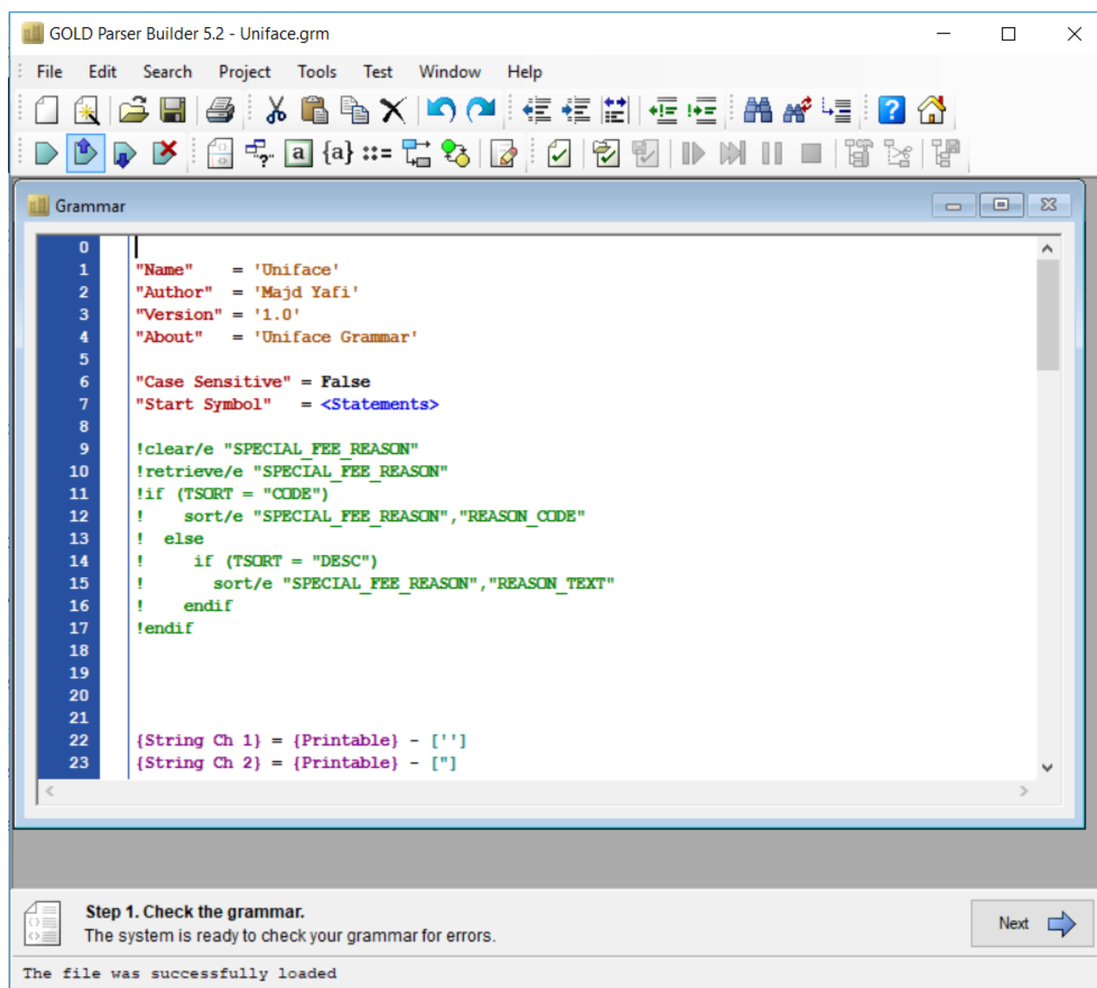

Figure 8.3: Gold Parser - Tool Main Screen with Example Code for Parsing Uniface

The main screen showing in figure 8.3 offers a simple editor featured with syntax highlighting for Gold Parser syntax.

A valid and complete grammar file is split up into two main logical parts. The first part is meta-data that defines the owner, author, file version and includes a short description of the language being built. This meta-data is ignored by the parser engine but is stored for reference. The second part is a block that defines the language attributes such as case sensitivity and the token which the grammar starts at. This token is the starting point for the generated DFA tree.

Some programming languages use indentation as tokens; these languages use the off-side rule (Landin [324]) to compile the syntax. Uniface does not adhere to these rules, therefore, Gold Parser was instructed to omit white spaces and tabs. These characters are called ‘noise’ and would be piped into the noise channel to be removed prior to generating the parse table.

The parsing table generated by the Gold Parser has been provided in appendix D.

8.3.7 Main Screen - User Interface

The main screen (figure 8.3) features a toolbar to access the other components that Gold Parser contains such as

- Test Window,
- DFA Screen,
- Search Window, and
- Editing functions.

A simple control panel is presented at the bottom of the screen to

- Control the flow of compiling the grammar,
- Save the Enhanced Grammar File,
- Generate the parse table,
- Correct warnings and errors, and
- Test sample input.

8.3.8 Analysis of Input

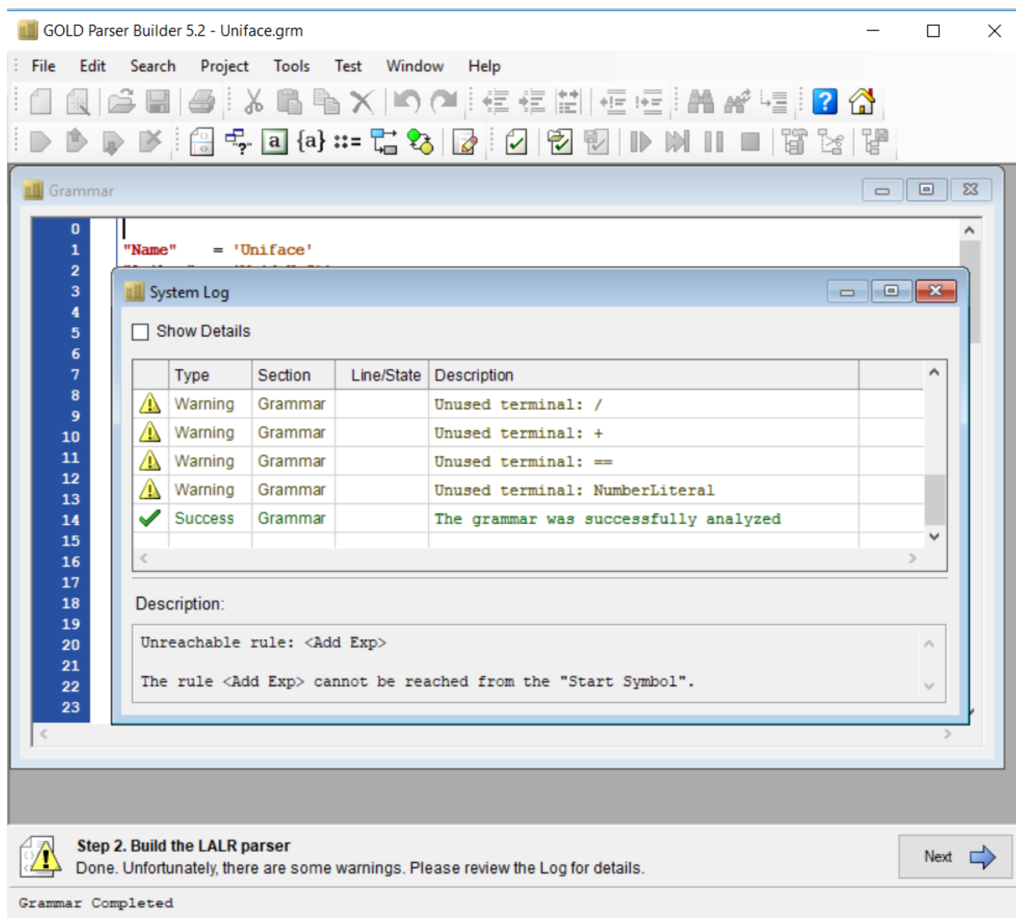

Figure 8.4: Gold Parser - Analyser Screen

Figure 8.4 shows the analyser screen which provides indicators regarding the correctness of the grammar syntax. These indicators are categorised into three categories of feedback:

- Red (Failure): Invalid syntax error is raised when Gold Parser fails to parse the syntax or when the grammar rules are inconsistent,
- Yellow Warning Triangle (Warning): inaccessible terminals generate warnings. An inaccessible terminal is a terminal for which there is no path that leads to it on the DFA tree for a given grammar,
- Green Tick (Success): A green tick indicator for success terminates the analyser list if the grammar does not contain errors.

8.3.9 Implementing Grammar Rules in Gold Parser

Gold Parser is built with two different components:

- the Builder, and
- the Engine,

Gold Parser can generate the parsing table, as shown in appendix E, in different programming languages without altering the process of writing and compiling grammar rules. Each is discussed below in turn.

Gold - Builder

The Builder is responsible for creating the DFA and LALR parsing tables. It uses the input grammar and processes it. The output is then saved to the CGT which can be used later, either with the Builder to amend it, or with the Engine for consumption. The Builder has the ability to generate a skeleton-program template which is used as the base to write programs that run on top of the input grammar.

Gold - Engine(s)

The engines are language and platform specific, therefore they vary greatly in terms of performance and implementation. This research has selected the *Calitha* engine that runs with C# .NET which can be evaluated with ANTLR code that is generated in the same language.

This engine, which has been also used by other researchers such as Buus et al. [325], Akbar et al. [326], Simões [327], generates a C# .NET skeleton program. The skeleton template contains the fundamental methods to read a stream of strings from the files which represents the CGT, and the input.

The following grammar represents parts of the grammar rules implemented in Gold Parser for this research:

```
1  "Name"    = 'Uniface'
2  "Author"  = 'Majd Yafi'
3  "Version" = '1.0'
4  "About"   = 'Uniface Grammar'
5
6  "Case Sensitive" = False
7  "Start Symbol"   = <Statements>
8
9
10 {String Ch 1} = {Printable} - ['']
11 {String Ch 2} = {Printable} - ["]
12
```

```
13 Id            = {Letter}{AlphaNumeric}*
14
15 ! String allows either single or double quotes
16
17 StringLiteral =  '"'  {String Ch 2}* '"'
18
19
20 NumberLiteral = {Digit}+('.'{Digit}+)?
21
22 <Statements> ::= <Statement> <Statements>
23                | <Statement>
24
25 <Statement>  ::=  <BuiltInStatement> | <If>
26  <BuiltInStatement>   ::= <Clear> | <Retrieve> | <Sort>
27 <Clear> ::= clear <ClearParam> StringLiteral
28 <ClearParam> ::= '/e' | <>
29
30 <Retrieve> ::= retrieve <RetrieveParam>   StringLiteral
31 <RetrieveParam> ::= '/e' | <>
32
33 <Sort> ::= sort <RetrieveParam>   <SortValue>
34 <SortValue>::= StringLiteral | StringLiteral ',' <SortValue>
35
36 <SortParam> ::= '/e' | <>
37
38 <If> ::= if '(' ID <Compare> StringLiteral ')' <Statements>  <Else> endif
39
40 <Else> ::= else <Statements> | <>
41
42 <Compare>::= '>' | '<'  | '<='  | '>=' | '<>' | '='
43
44 <Expression>   ::= <Expression> '>'  <Add Exp>
45                  | <Expression> '<'  <Add Exp>
46                  | <Expression> '<=' <Add Exp>
47                  | <Expression> '>=' <Add Exp>
48                  | <Expression> '==' <Add Exp>
49                  | <Expression> '<>' <Add Exp>
50                  | <Add Exp>
51
```

```
52  <Add Exp>       ::= <Add Exp> '+' <Mult Exp>
53                    | <Add Exp> '-' <Mult Exp>
54                    | <Add Exp> '&' <Mult Exp>
55                    | <Mult Exp>
56
57  <Mult Exp>     ::= <Mult Exp> '*' <Negate Exp>
58                   | <Mult Exp> '/' <Negate Exp>
59                   | <Negate Exp>
60
61  <Negate Exp>   ::= '-' <Value>
62                   | <Value>
63
64  <Value>        ::= ID
65                   | StringLiteral
66                   | NumberLiteral
67                   | '(' <Expression> ')'
```

8.3.10 Producing a Derivative Tree

BNF or EBNF are commonly used to draw context-free grammar rules. The language is compressed into a single node called the 'start symbol'. Parse trees (also termed 'derivation trees') can also depict the hierarchy where the start symbol becomes the root of the tree. The parser tree/derivative tree is the result of sorting tokens according to grammar semantic and syntactical rules.

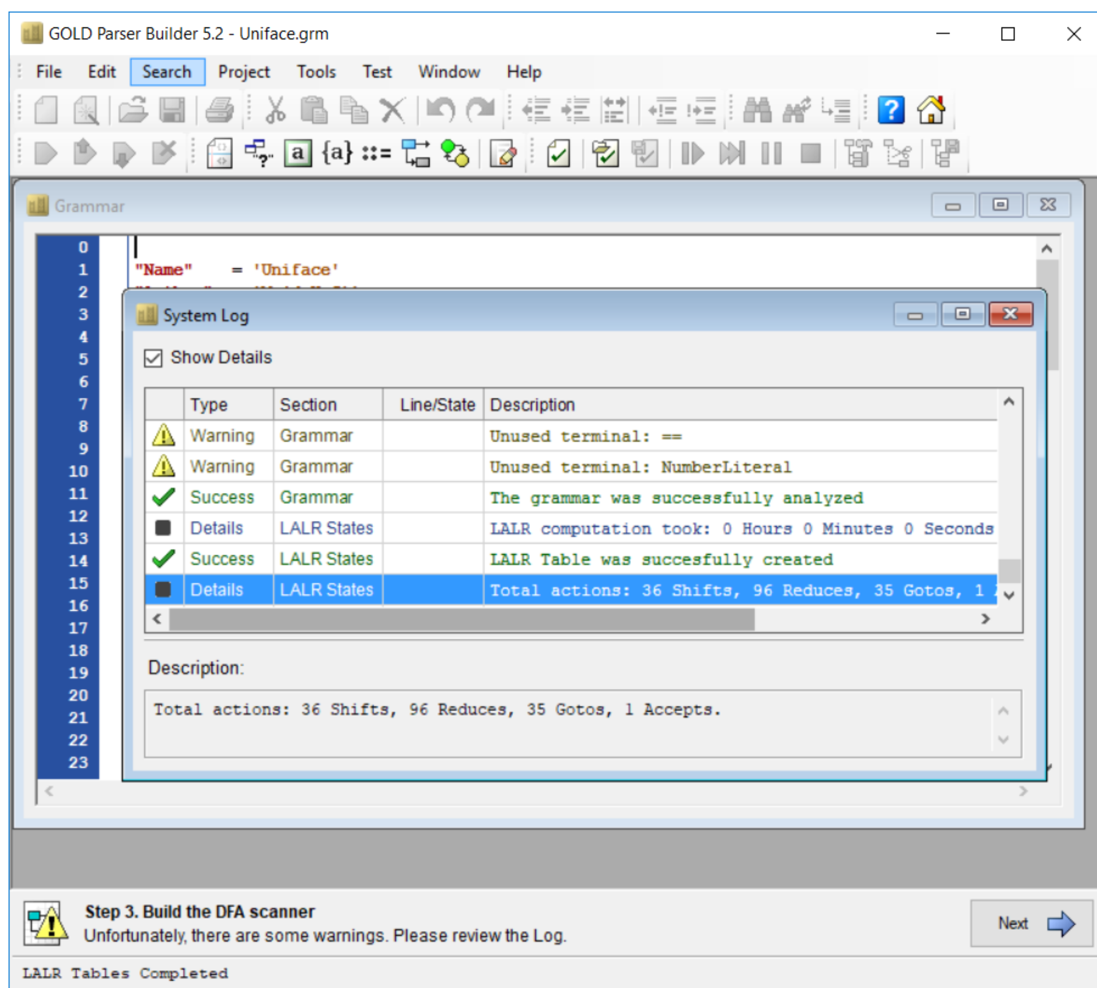

Figure 8.5: Gold Parser - LALR Table Generation Screen

The Left-to-Right (LR) derivation technique identifies the completed rules using tables as shown in the bottom three rows of figure 8.5.

The same figure shows that the LALR table was generated. The table replaces grammar rules for programs that would alternatively use grammar rules.

It skips over the stream of strings for the character when a rule has not been matched. LR also reduces sub-strings to non-terminals at compile time. This is beneficial because it both reduces the number of rules and also increases the level of abstraction, a fundamental concept for construction of derivation trees and traversing them.

8.3.11 Testing Grammar

Testing a sample grammar is feasible post-generation of the LALR table. The test window (figure 8.6) can be accessed either through the toolbar menu or the wizard at the bottom of the screen.

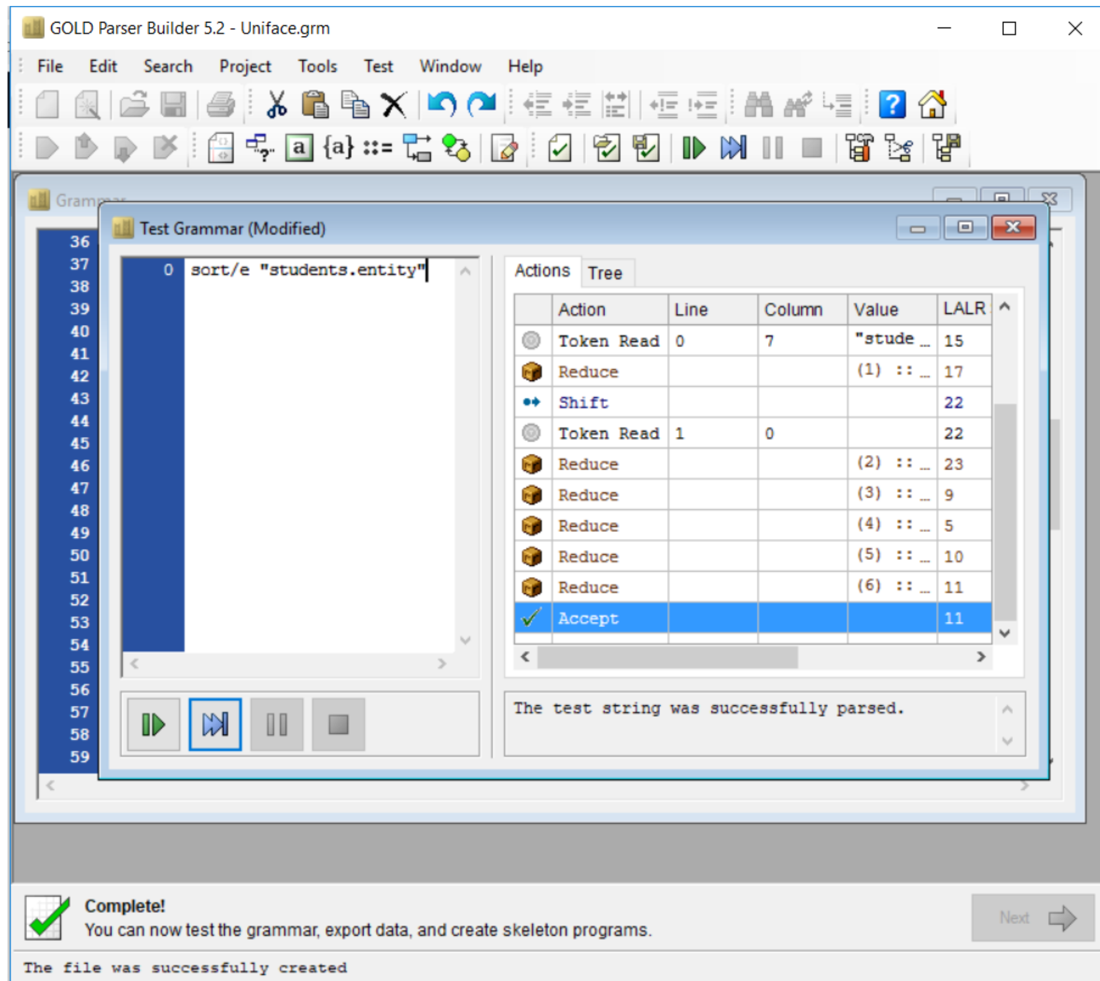

Figure 8.6: Gold Parser - Test Window

Figure 8.6 shows the test window in action. The test window is split into two panels. The left panel is used to load data. In this example the Uniface sort method was called and validated against the written grammar. Below the editor, are step by step controls to monitor how Gold Parser interacts with the input and to show the current DFA state.

If the test passes all its grammar rules, a success pop-up is displayed, as shown in figure 8.7 and a parse tree is composed.

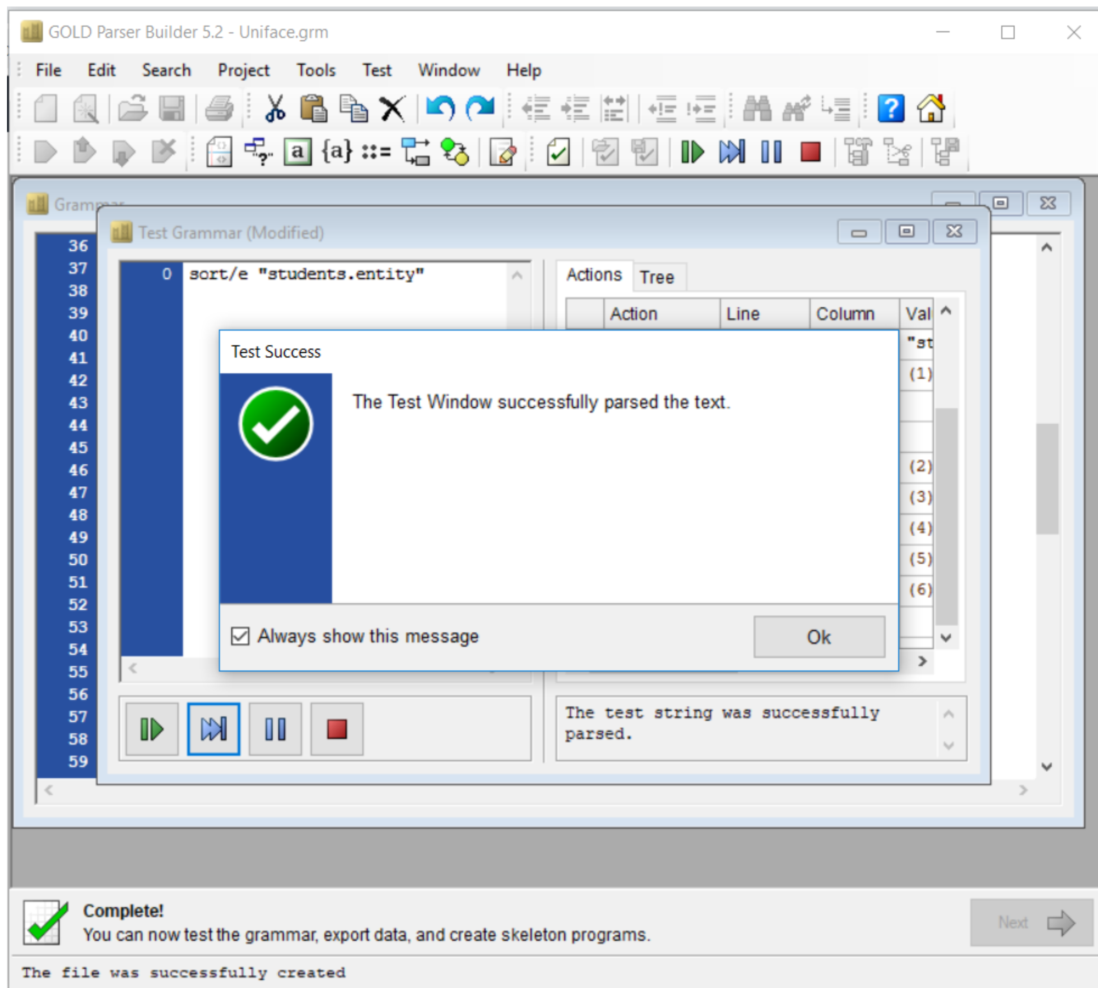

Figure 8.7: Gold Parser - Test Window - Input Sample Parsed Successfully

Success in this step is crucial to eliminate any doubts about the correctness of the grammar syntax. The pop-up in itself is less relevant than the assurance it gives regarding the grammar syntax correctness, and the parser it generated.

8.3.12 Gold Parser Usage in this Research

The Gold Parser tool is a Windows specific tool that runs on 32-bit operating systems. For this research Windows 10 was used to download and install the utility. The selected engine was C# and the code was written using Visual Studio 2017 IDE.

The next section 8.4 illustrates a mechanism to draw Uniface language rules in EBNF and translate the syntax using ANTLR.

8.4 Implementation in ANTLR

This section shows how Uniface grammar rules were implemented and debugged. It also shows how the implementation results were output in ANTLR.

8.4.1 ANTLR Parser

ANTLR was built using the Java programming language. Therefore, having hardware and software able to run a *Java Virtual Machine* is a prerequisite for installing and running ANTLR. However, ANTLR's output is not restricted to Java code as it can be used to generate parsers in C++ or C#. The ANTLR library itself is available to other external tools which invoke it, as ANTLR is operating system neutral. The installation and use could differ between the platforms on which it is installed.

In this research, a Windows 10 platform was used. This needed a new environment variable $CLASSPATH$ to be added, pointing to the physical location of the library.

8.4.2 ANTLR Parser Basic Example

The most basic example, Hello, is provided by the author of ANTLR which tests the library output, as shown below:

```
1 grammar Hello; // Define a grammar called Hello
2 r : 'hello' ID ; // match keyword hello followed by an identifier
3 ID : [a-z]+ ; // match lower-case identifiers
4 WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines, \r (Windows)
```

ANTLR 'recognisers' can generate a wide range of error messages for invalid syntax. Recognisers are designed to minimise the number of generated errors per syntax line. In addition, recognisers handle common errors efficiently. Common errors as described in Parr [188] are mismatched token, no viable alternative, and early exit from an EBNF (...)+ subrule loop.

Running ANTLR using the above example should output a binary executable files of the recogniser.

In the code example (section 8.4.4) the first line with the keyword 'grammar' sets the grammar name variable, in the example it is set to 'Uniface'.

Two output files are generated and named as {grammar name}Parser and {grammar name}Lexer. If the target language is not explicitly specified, ANTLR will default to generating Java files. ANTLR also provides a tool that uses Java reflection to traverse the code and generate an AST with a graphical interface to illustrate the generated AST. This tool is used to inspect the generated tokeniser (the component in ANTLR that generated tokens of the input) and confirm results.

8.4.3 Parser Backtracking

The parsing part of the back-end (Data) layer of the proof of concept system, has been implemented on two different parser generator platforms: ANTLR and Gold Parser.

The former is a public-domain parser generator that uses $LL(K)$ grammar and EBNF grammar syntax. ANTLR employs the concept of predicates which provide semantic directives to the parser. In addition, ANTLR can generate an AST automatically using the development tools that come with it, Parr and Quorg, 1995 [83]. ANTLR uses backtracking which is an expensive technique; in fact, backtracking is an exponentially complex problem. The recogniser traverses a tree of decisions and when parsing rules, it will try alternatives until the decision is made, Parr, 2013 [188].

8.4.4 Implementing Grammar Rules in ANTLR

In this research the target was Uniface. Therefore, the researcher extended ANTLR grammar to reflect Uniface syntactical formation. Uniface grammar for the extract of Uniface code input, shown in appendix B, looks like the following:

```
1  grammar Uniface;
2
3  u: ENTRY NAME statements END;
4
5  statements: statement*;
6  statement:
7          functionRule
8          | assignemntStatement
9          | ifRule
10         | expr
11         ;
12
13 assignemntStatement:
14         VARIABLE '=' expr
15         ;
16
17 ifRule:
18         IF LEFTBRACKET expr RIGHTBRACKET
19                 statements
20                 elseifRule*
21                 elseRule?
22         ENDIF
23         ;
```

```
24
25  elseifRule:
26          ELSEIF LEFTBRACKET expr RIGHTBRACKET statements
27          ;
28
29  elseRule: ELSE statements ;
30
31  expr
32          : expr '.'
            ↪  expr                                                      #
            ↪  MemberDotExpression
33          | '++'
            ↪  expr
            ↪  PreIncrementExpression
34          | '--'
            ↪  expr
            ↪  PreDecreaseExpression
35          | '+'
            ↪  expr
            ↪  UnaryPlusExpression
36          | '-'
            ↪  expr
            ↪  UnaryMinusExpression
37          | '~'
            ↪  expr
            ↪  BitNotExpression
38          | '!'
            ↪  expr
            ↪  NotExpression
39          | expr'[' NUMBERS ':' NUMBERS
            ↪  ']'                                   # SliceExpression
40          | expr '&lt;'
            ↪  expr                                                                  #
            ↪  LessThanExpression
41          | expr '&gt;'
            ↪  expr                                                                  #
            ↪  GreaterExpression
```

```
42          | expr '*'
            ↪  expr
            ↪  MultiplyExpression
43          | expr '/'
            ↪  expr
            ↪  DivideExpression
44          | expr '%'
            ↪  expr
            ↪  ModulusExpression
45          | expr '&amp;'
            ↪  expr                                                              #
            ↪  AddExpression
46          | expr '|'
            ↪  expr
            ↪  OrExpression
47          | expr '+'
            ↪  expr
            ↪  AddExpression
48          | expr '-'
            ↪  expr
            ↪  SubtractExpression
49          | expr '='
            ↪  expr
            ↪  EqualsExpression
50          | expr '!='
            ↪  expr                                                              #
            ↪  NotEqualsExpression
51          |
            ↪  literal
            ↪  LiteralExpression
52          | '(' expr
            ↪  ')'
            ↪  ParenthesizedExpression
53          ;
54
55  literal: NAME | VARIABLE | NUMBERS | STRING | BOOLEAN;
56
57
58  functionRule: clearFunction | retrieveFunction | sortFunction;
```

```
59
60
61
62 clearFunction: CLEAR FUNCARG? STRING;
63 sortFunction:
64         SORT FUNCARG? STRING (',' expr)*;
65 retrieveFunction: RETRIEVE FUNCARG? STRING;
66
67
68 OPERATOR:
69         ANDOPERATOR
70         | ORPERATOR
71         | EQUALOPERATOR
72         | STAROPERATOR
73         | LESSTHANOPERATOR
74         | GREATERTHANOPERATOR;
75 ANDOPERATOR: '&amp;' | '&uAND;'; //| '&uAND;'
76 ORPERATOR: '&uOR;' | '|';
77 EQUALOPERATOR: '&uEQ;' | '=';
78 STAROPERATOR: '&uALL;' | '*';
79 LESSTHANOPERATOR: '&lt;' | '<';
80 GREATORTHANOPERATOR: '&gt;' | '>';
81
82 SINGLE_LINE_COMMENT: ';' ~[\r\n]* -> skip;
83
84 ENTRY: 'entry';
85 END: 'end';
86 IF: 'if';
87 ELSEIF: 'elseif';
88 ELSE: 'else';
89 ENDIF: 'endif';
90 BOOLEAN: 'true' | 'false';
91 FUNCARG: '/e';
92 CLEAR: 'clear';
93 RETRIEVE: 'retrieve';
94 SORT: 'sort';
95 COMMA: ',';
96 VARIABLE: '$'+ (NAME | NUMBERS);
97
```

```
98  //NUMBERS : [1-9][0-9]*;
99  NUMBERS: [0-9]+;
100 NAME: [a-zA-Z_]+ (NUMBERS)?;
101
102 STRING:
103         '"' (~["\\\r\n\u0085\u2028\u2029] | NAME)* '"'; // '"' (~'"'|'"')*
        ↪  '"'
104
105 //KEYWORDS: 'if' | 'endif' ;
106
107 LEFTBRACKET: '(';
108 RIGHTBRACKET: ')';
109
110 WS: ( ' ' | '\t' | '\r' | '\n') -> skip;
```

Figure 8.8: ANTLR: String Rule Railroad Representation

In the above example (Line 102), strings are defined in such a way as to read all characters embodied in double quotes with a skip for paragraph separators and new line special characters (as also shown in railroad diagram form in Figure 8.8).

Another example is the “If-Rule” as shown in the code above between lines 17 and 29 inclusive.

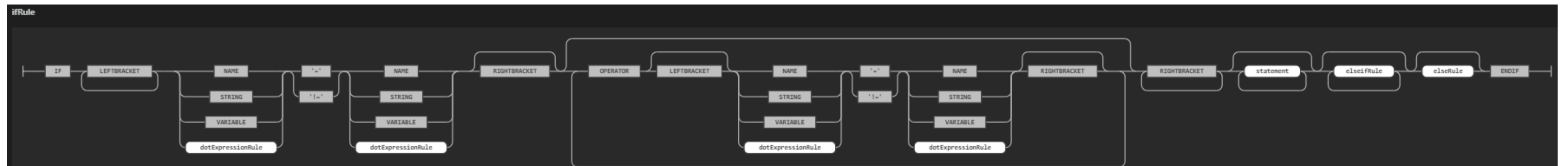

Figure 8.9: ANTLR: If-Rule Railroad Representation

The If-Rule (for its Railroad representation see figure 8.9), is a complicated rule as it supports recursion of expressions. Each expression can have a binary expression that evaluates the overall condition value to be either “true” or “false”. Uniface also supports nested ‘If’ conditions where each might have an unlimited number of ‘Else if’ statements and/or a terminating else statement. Each statement's implementation can either be an atomic statement or a list of statements which the ANTLR tokeniser will recognise using defined grammar rules.

The structure of an If-statement in Uniface, is defined below. Each regular expression in square brackets ‘[’ and ‘]’ represents separate valid routes through the if statement.

```
1 if [(]+ [condition]* [[binary operator] [condition]]* [)]+
2     [statements]*
3 [[elseif]
4     [statements]*
5 ]*
6 [else]?
7     [statements]*
8 endif
```

The generated parser has an array of Deterministic Finite Automation (DFA) type which determines the decision for the next state. The parser contains references to the operators, literals, and rules.

8.4.5 Debugging ANTLR Output

ANTLR provides some tips, Parr [188], when compiling the output, which helps in debugging the input. For example, [@1,6:13='retrieve',<2>,1:6] indicates that that the term has appeared twice in the context, starting from character 6 and ending with 13. The index starts at 0. The token string holds the value of ‘retrieve’ which is a keyword in Uniface. In the given context the keyword is being identified as an ID. The term is located on line 1 where line indexing starts from 1. The term is located at column (character) 6 starting from index 0.

8.4.6 ANTLR ‘Listener’ Output File

ANTLR also generates an interface called {Grammar Name}Listener. The interface contains method signatures that get invoked either when the relevant rule is invoked, or its execution is completed.

The code fragment below is from generated C# code from the listener file:

```
1 /// <summary>
2 /// Enter a parse tree produced by <see cref="HelloParser.ifRule"/>.
3 /// </summary>
```

```
4 /// <param name="context">The parse tree.</param>
5 void EnterIfRule([NotNull] HelloParser.IfRuleContext context);
6 /// <summary>
7 /// Exit a parse tree produced by <see cref="HelloParser.ifRule"/>.
8 /// </summary>
9 /// <param name="context">The parse tree.</param>
10 void ExitIfRule([NotNull] HelloParser.IfRuleContext context);
```

8.4.7 Augmented Transition Network (ATN) Visualisation

ATN is a data-structure of nodes and arcs, where the nodes correspond to states in finite state machine and the arcs represent a transition from a state to state, according to Woods et al. [328].

The ANTLR lexer contains a serialised object that represents the grammar Augmented Transition Network (ATN). ATNs can be visualised as shown in figure 8.10 below which illustrates a String-ATN.

An ATN is built on top of a Markov Chain model, Snell [329] Finite State Machine (FSM) and is used to represent the structural and operational definition of formal languages. It extends a Markov Chain by adding recursion to it. The benefits of an ATN are in increasing efficiency and improving ambiguity detection. Figure 8.11 shows the ATN diagram for the If-Rule. In that figure, ϵ refers to edges of an inexpensive path which the parser can follow for a next valid state without the need to consume input syntax. Edges with other labels denote the path(s) which the parser can take to change its state by reading the next token from the input. ATN are used to debug and optimise parsers as well as visualising their state and decision graph.

The ATNs have been generated by the author. Internally they used a declarative language called 'dot'. According to Koutsofios and North [330] "dot draws directed graphs as hierarchies".

8.4.8 Call Graph Visualisation

To demonstrate the number of calls which are made to a specific rule and calls which invoke the rule, a call graph was generated for the String-rule, as shown in figure 8.12. Call graphs can help software developers to inspect the calls which are made between routines and subroutines. Visualising the calls can increase the level of a software developer's understanding of the possible behaviours of the application.

8.4.9 ANTLR 'Base Visitor' Partial Class

ANTLR generates a Visitor class named {grammar name}BaseVisitor which is generated as a partial class in the Microsoft .NET language. This partial class contains the methods that are triggered when a rule is visited. For example the If-Rule Visitor looks like the following in the generated C# code:

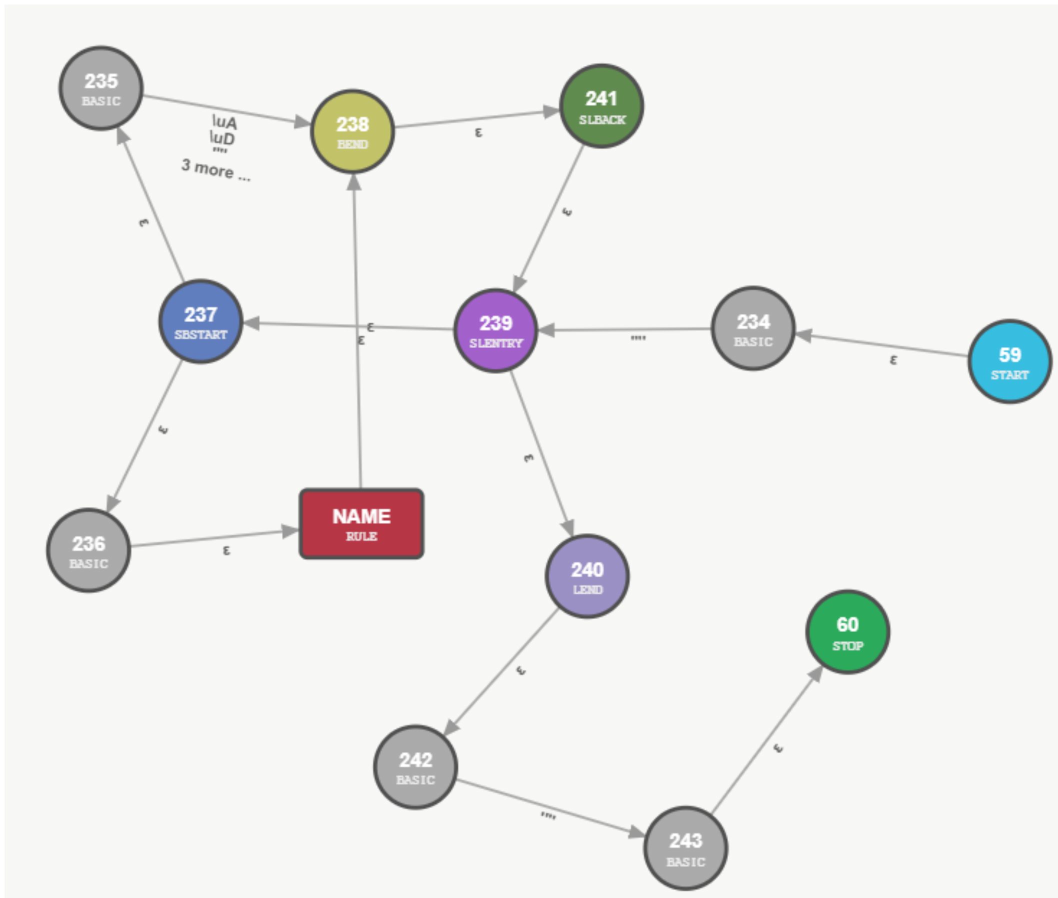

Figure 8.10: String-Rule ATN Visualisation

```
1  public virtual Result VisitIfRule([NotNull] HelloParser.IfRuleContext
   ↪   context)
2  {
3      return VisitChildren(context);
4  }
```

8.4.10 Using ANTLR Output

The lexer and parser which had been generated as a result of compiling Uniface grammar rules in ANTLR, were used as input to the custom program written by the author to utilise the grammar rules and traverse its AST.

This program utilised the 'Visitor' design pattern to traverse and translate the syntax of the Uniface code and extend it to translate the parsed tokens into legitimate C# tokens.

The call to the reader streams (as shown in line 2 in the snippet below) inputs as a series of string

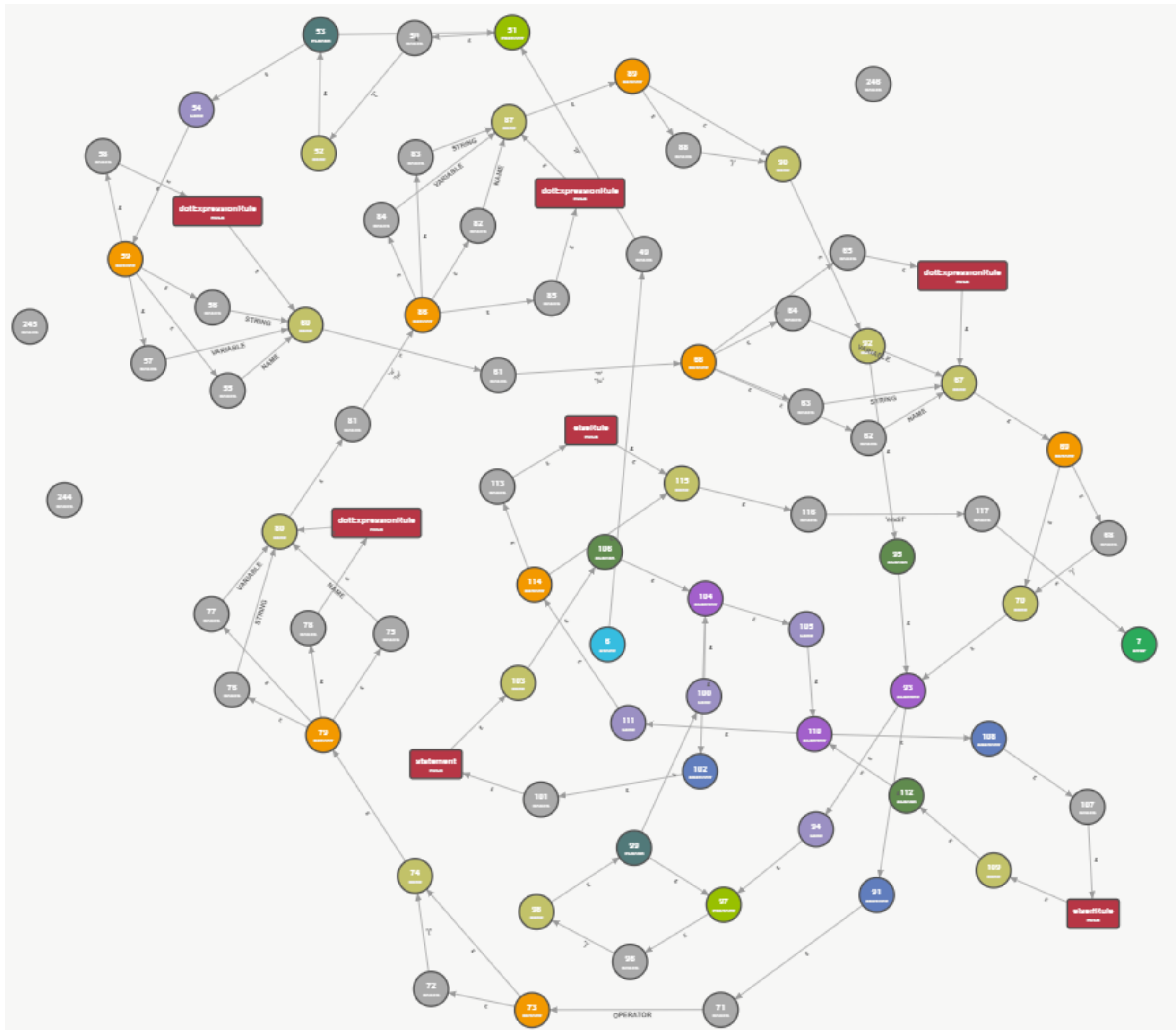

Figure 8.11: If-Rule ATN Visualisation

characters. Then the lexer performed a lexical analysis whereby the characters were scanned and converted into sequence tokens. The parser then formed a parsing tree based on the input from the lexer. To handle unexpected errors the routine *AddErrorListener* was used, which invokes a user-defined routine that handles the triggered error(s).

A sample snippet of code is shown below:

```
string input = File.ReadAllText("input.txt");
AntlrInputStream inputStream = new AntlrInputStream(input);
UnifaceLexer lexer = new UnifaceLexer(inputStream);
```

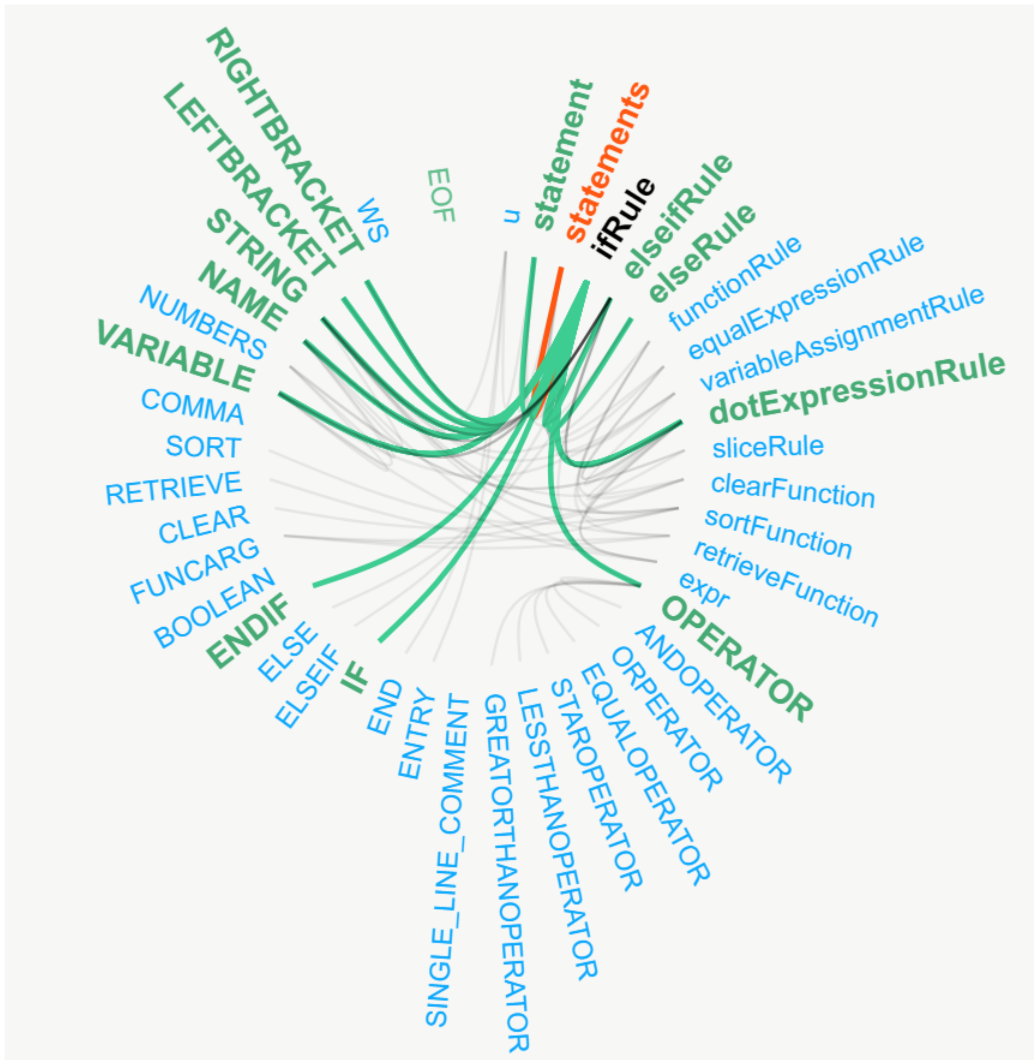

Figure 8.12: String-Rule Call Graph Visualisation

```
4  CommonTokenStream commonTokenStream = new CommonTokenStream(lexer);
5  UnifaceParser speakParser = new UnifaceParser(commonTokenStream);
6  speakParser.AddErrorListener(new Program());
7  var ast = speakParser.u();
8  MyVisitor2 visitor = new MyVisitor2();
9   visitor.Visit(ast);
```

8.4.11 Traversing Grammar Rules

Software engineering design patterns were utilised in order to visit each rule and translate it. Design patterns are solutions to common problems encountered in the software engineering field. The UML design in figure 8.13, taken from W3s Design, shows classes (objects meta-data) and the relationship across them and shows how objects interact with each other.

A visitor design pattern for instance, separates the logic from the class's design structure. The object logic can become extendable without changing the object formation. This can be achieved by adding virtual functions to an object and allows the implementer to override the function's logic.

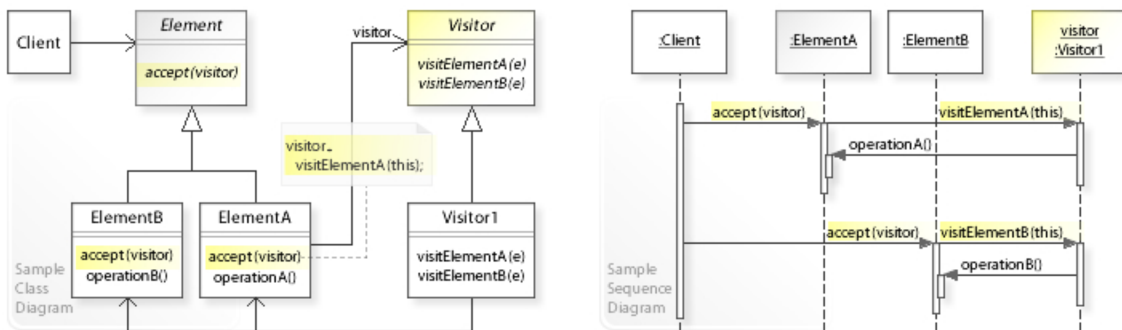

Figure 8.13: UML Representation of the Visitor Design Pattern. Taken from W3s Design [11]

A formal representation of the visitor design pattern, is illustrated in Figure 8.14 taken from Alzahrani, Yafi et al. [12] using the LePUS3 tool.

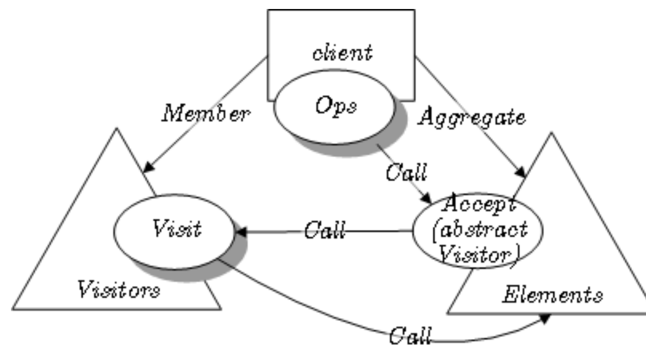

Figure 8.14: LePUS3 Representation of Visitor Design Pattern From Alzahrani, M. Yafi et al. [12]

In this research, ANTLR and Gold Parser both utilised the visitor design patterns to visit the rules. The code in appendix C shows one way of visiting the rules. The *StringBuilder* object in

.NET was used to gather information about the object being visited and its child objects as they exist in the parsed syntax tree.

8.5 Comparison Between Gold Parser and ANTLR

The parsing part of the back-end (Data) layer of the proof of concept system, has been implemented on two different parser generator platforms: ANTLR and Gold Parser (first discussed in sections 3.4.6.1 and 3.4.6.3 respectively).

This research has used two different parser generators aiming at generating a compiler based on formal grammar input to transform and transpile Uniface syntax into object orient language syntax, being C#. The C# output (as produced by ANTLR) is provided as appendix C.

The two parsers use two different parsing techniques. ANTLR uses $ALL(*)$, also known as Adaptive LL(*) compared to Gold Parser which uses $LALR$, also known as Look Ahead LR.

This research has aimed to show that going bottom-up with transforming 4GLs is feasible either way. However, the semantics might be contextually different.

ANTLR has the advantage that ALL(*) offers, which is dynamic analysis of grammar at run-time. Static analysis has to anticipate the indefinite patterns for unknown input. ANTLR dynamically analyses the given input at run-time.

Gold Parser, as shown in section 8.3 and ANTLR tools each offer a user interface to build and test the written grammar. Gold Parser generates LALR and DFA parsing tables separate from the logic that contains the code for the parsing algorithms. Gold Parser usage and operation was explained in section 8.3.

8.6 Evaluation of Gold Parser and ANTLR Algorithm Strategy

Parr et al., 2014 [189] said that

> "Computer language parsing is still not a solved problem in practice, despite the sophistication of modern parsing strategies and long history of academic study."

One of the parsing issues that Parr et al., 2014 referred to was that performance complexity can be unpredictable.

The complexity measure for each language parsed and transformed can best be described using Big-O notation according to Chivers and Sleightholme [331].

ALL(*) as used by ANTLR parser in this research has a complexity measure of $O(n^4)$ but it is claimed to be linear in practice by Parr et al.

The ALL(*) algorithm provided improved performance over other parsing strategies (such as GLL $O(n^3)$ and GLR $O(n^{p+1})$ where n is the number of grammar rules and p is the length of the longest production in the grammar rules). By moving the grammar analysis to include it in the input 'parse-time' phase. This allowed ALL(*) to handle non-left-recursive grammar and also increased the performance of ALL(*) by ~135 times over GLL and GLR (as claimed by Parr et al., 2014).

Other strategies such as LL and LR (top-down and bottom-up respectively) consume resources using non-deterministic strategies (such as the LALR algorithm used in this research with Gold Parser). Non-deterministic strategies suffer from unpredictable run times and could return a forest (multiple trees) when parsing. When parsing computer languages this can lead to ambiguity which results in parsing errors.

ALL(*) uses dynamic programming to memoise the results of sub-parsers. Sub-parsers are created at the decision point where each sub-parser continues to parse the rest of input. The sub-parser is dumped if it fails to parse the remaining input. If two sub-parsers succeeded in parsing the remaining input, the one with the minimum production number would take over.

ALL(*) memoisation builds a cache of DFAs represented by a Graph-Structured Stack (GSS), Tomita [332]. This technique performs better than the exponential performance that is usually observed with non-deterministic strategies.

Therefore, the choice of using ANTLR is recommended for this research both for its performance features and also for its strategies to overcome grammar rules ambiguities.

8.7 Conclusion

This chapter has covered:

- S.O.L.I.D. Principles (section 8.1),
- .NET Framework and C# Programming Language (section 8.2),
- Implementation in Gold Parser (section 8.3),
- Implementation in ANTLR (section 8.4),
- Comparison Between Gold Parser and ANTLR (section 8.5),

being the languages and tools used to take code from the XML-like code as output by Uniface, into a parsed form output as C# code (using the XSD Scheme described in section 5.2).

Lastly, in section 8.6, the chapter discussed two parsers in terms of complexity using Big-O notation.

Chapter 9 (Extending from Uniface to Other 4GLs) takes the principles applied specifically to

the proprietary Uniface 4GL in the previous chapters, and applies these both to some other 4GLs, and also to a Mini-4GL as created by the author.

Chapter 9

Extending from Uniface to Other 4GLs

The previous discussion (chapter 3 to chapter 8) has dealt with applying the author's novel methodology to a single proprietary example 4GL (Uniface), on a single system (ESIS). Proof of concept implementation that such an approach was viable, has been provided in chapter 10 Results, section 10.4.1.

This chapter has now brought the same approach to bear (although discussed in less detail) firstly for two other 4GLs (Informix and Apex) (section 9.1), and secondly on a Mini-4GL of the author's own invention (sections 9.2 and 9.3).

The syntax of each of these three languages have only been reverse engineered only to a very limited extent - just enough to determine broadly whether the author's method appears to work on languages other than Uniface which was itself studied in more detail, although not fully.

The Mini-4GL is one which the author originally built as part of more general research (stepping stones) into 4GLs and how they were built. It was only later that the same methodology was applied to demonstrate if it also could be successfully reverse engineered using the same novel approach as had been applied to Uniface.

Because the Mini-4GL is entirely new, the discussion first needs to cover the forward engineering process used to build a new language (section 9.2), before going on to discuss the application of the author's syntactical reverse engineering approach, based on a very small exemplar original 4GL system (section 9.3). The latter acted as a useful mini test bed because its grammar rules were exactly known - having been created on a bespoke basis.

The forward engineering process to build a new language is a desirable precursor to reverse-engineer existing ones, Baxter and Mehlich [333]. This language defined keywords and arguments

with custom data-types.

9.1 Reverse Engineering Other 4GLs

The 4GLs covered in this section were chosen as their specifications were readily available. Although they were not model-driven XML-like 4GLs, like Uniface, it was felt that even a very small scale trial would be a useful demonstration that the author's method and methodology would also apply to them, to provide syntactical translation.

A demonstration that the methodology outlined in this thesis was also potentially viable on a different model-driven XML-like 4GL than Uniface, has been given by using the method on the author's Mini-4GL in section 9.3 (Reverse Engineering the New Mini-4GL Language).

It was not found possible to demonstrate the author's methodology on other mainstream model-driven XML-like 4GLs as they were proprietary and also were unavailable to the author.

The transformation process for all the 4GLs covered in this chapter included the parsing step and code-generation that is based on traversing the AST by means of the 'Visitor' design pattern.

The number of terminal and non-terminal nodes varied for each of the 4GLs presented in this thesis.

The steps followed to reverse engineer the Mini-4GL (section 9.2), as well as for the other 4GLs, are equivalent to those used with Uniface. Chapter 8 discussed the implementation phases of Uniface grammar rules using both Gold Parser and ANTLR. Only ANTLR parser was used for the other 4GLs and Mini-4GL.

These steps can be summarised as follows:

- Implementing the grammar rules as shown in sections 8.3.9, and 8.4.4,
- Generating the parse table as shown in sections 8.3.6, and 8.3.10,
- Testing the grammar (section 8.3.11) and debugging the output as shown in sections 8.4.5 and 8.4.10,
- Traversing the AST and applying the custom transformation logic as shown in section 8.4.11.

A successful outcome to produce viable C# code, was achieved for each of the other 4GL languages, as shown in appendix E.

Fuller results comparing the abstract syntax trees (in the form of chunking trees) for the originating source languages, and the out language for each of the other 4GLs, as far as the grammar rules that were created, have been given for Informix in section 10.4.2 and for Apex in section 10.4.3.

This research extended its methodology and proof of concept to target other 4GLs that share some similar characteristics to Uniface. The selected languages both are 4GLs and operate on top of an interpreter to execute their commands. In addition, both of these 4GLs are capable of storing business logic within.

The author's method of reverse engineering the syntax of proprietary 4GLs has had to be adjusted for each 4GL tackled since they each exhibit different characteristics.

9.1.1 Informix

Informix syntax drives the Informix database engine.

Informix syntax has been well documented using Railroad diagrams which are accepted as a formal representation of the language syntax and can be found in Informix's own documentation [13].

The existence of a formal representation of Informix syntax simplified the process of generating its relevant grammar rules. The selected snippet of code for translation has been shown in section 10.4.2. It was just a simple snippet of code and would not be able to run queries to extract data from an Informix database system.

In the case of Informix the author's work did not cover the area of running an Informix query and the code that runs between the 4GL and its underlying engine. A more in depth analysis has been raised as a topic for future researchers, as dealt with in section 11.6.

9.1.2 Apex

Apex was syntactically more similar to object-oriented languages than Informix. Apex runs and drives the 'SalesForce' Customer Relationship Management (CRM) system.

Apex can be easier to translate given its syntactical similarity to the target language C#. even more, Apex is less coupled to the underlying engine than Informix which would reduce the size of output and keep it closer to the input source. A summary of results for Apex has been presented in section 10.4.3.

9.2 Building a New Mini-4GL Language

The rudimentary system named 'PharmacyX' was built using the author's own Mini-4GL language, and was designed using the Meta-Programming System (MPS) workbench system (Campagne [334], Völter and Pech, 2012 [335], Völter, 2011 [336]).

The PharmacyX Mini-4GL system had a very limited set of keywords and capabilities. It enabled developers using the Mini-4GL language to list an item names and the quantity of each item, and to output a valid HTML document. The document could be styled with custom cascading

style sheets (CSS). Therefore, the language defined a simplified version of the CSS language and reconstructed CSS grammar rules as well as adopting the implementation rules for HTML.

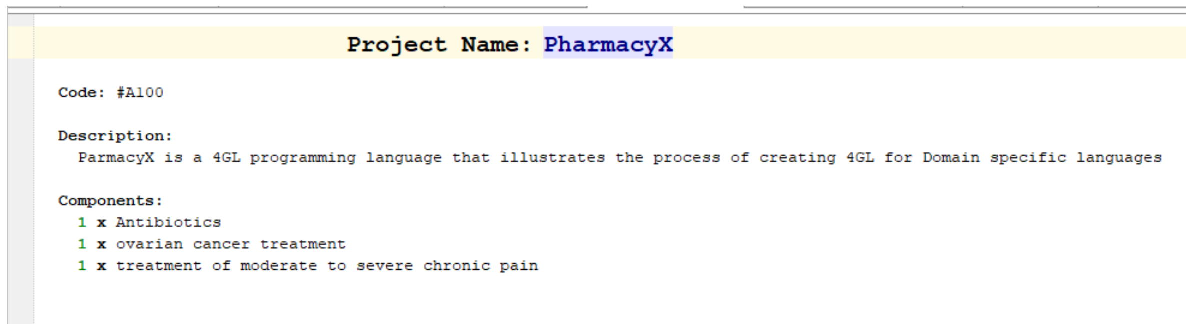

Figure 9.1: MPS Mini-4GL Language Editor

The document displayed as figure 9.1 defines the use of the keywords *Code*, *Description*, and *Components*.

The 'Code' keyword uses its argument to generate an identifier. It accepts an argument of the following regular expression.

```
[@|#][A-Z][0-9][0-9][0-9]
```

'Description' keyword takes a string argument and use it to display a short text description of the project.

The 'Component' keyword expects an array of an integer followed by the character 'x' and then a 'material'. The material object represents a pre-defined list of materials.

MPS reads the input, performs a static analysis on the language syntax at compile-time and validates the syntax against the language grammar rules as shown in figure 9.2.

MPS language workbench offers a grammar editor to define the new language grammar. Figure 9.2 illustrates the grammar rules editor.

```
<default> editor for concept
  node cell layout:
    [-
      Project Name: { name }
        <constant>
      Code: { code }
      <constant>
      Description:
        (- % description % /empty cell: <default> -)
        <constant>
      Components:
        (/ % components %          /)
          /empty cell: <default>
    -]

  inspected cell layout:
    <choose cell model>
```

Figure 9.2: MPS Grammar Rules Editor - The Grammar for the Main Screen of Mini-4GL

Figure 9.2 illustrate grammar rules for the language. These rules instructs MPS to generate an abstract syntax tree for the input code.

Each

- language token, and
- keywords such as: ‘code’, ‘description’, and ‘components’

are defined and implemented in a similar fashion. Non-terminals like ‘components’ can be interlinked to another non-terminals like ‘Project_Component’. ‘Project_Component’ is linked to the terminal ‘quantity’ as defined in figure 9.3.

```
<default> editor for concept Project_Component
  node cell layout:
    [- { quantity } x ( % component % -> { name } ) -]

  inspected cell layout:
    <choose cell model>
```

Figure 9.3: MPS Grammar Rules Editor Project_Component

'MPS Concepts' are the container (logical groups) of the grammar rules shown in Figures 9.2, and 9.3. 'Concepts' are extendable at run-time to deliver intended behaviour.

The Mini-4GL project had also implemented within it, a MPS Concept named 'Garage_Project' which implemented grammar rules for materials and also contained routines ('getExpenses' and 'getPrice') to calculate the cost (named expenses) and price of each item. Figure 9.4 illustrates the definition of 'getExpenses'.

```
concept behavior {

  constructor {
    <no statements>
  }

  public int getExpenses() {
    int expenses = 0;
    for (node<Project_Component> component : this.components) {
      expenses += component.component.price * component.quantity;
    }
    return expenses;
  }

  public int getPrice() {
    getExpenses() * 2;
  }
```

Figure 9.4: MPS Behavioural Editor for Mini-4GL

The 'TextGen' language feature within MPS, defines a model to text transformation", Makarkin [337]. 'TextGen' is used in this example to transform the nodes in the abstract syntax tree into text. The generated text represents the output which is the HTML file and associated CSS.

An example of 'TextGen' code is provided in figure 9.5 which represents the skeleton of the generated HTML file.

```
text gen component for concept HtmlFile {
file name : <Node.name>
file path : <model/qualified/name>
extension : (node)->string {
  "html";
}
encoding : utf-8
text layout : <no layout>
context objects : << ... >>

  (node)->void {
    append ${node.document};
  }
}
```

Figure 9.5: TextGen for the HTML Document

It directs MPS to generate a documents that adheres to the ‘Concept’ named ‘XmlDocument’ which is defined in the code as set out in Figure 9.6.

```
concept HtmlFile extends    BaseConcept
                 implements INamedConcept

  instance can be root: true
  alias: html file
  short description: <no short description>

  properties:
  << ... >>

  children:
  document : XmlDocument[1]

  references:
  << ... >>
```

Figure 9.6: The ‘Concept’ that Represents the Standard HTML Structure

The ‘Transformation menu’ language (within MPS) is used to define transformation menus that describe a hierarchical structure of sub-menus and actions that will appear in various locations in the Mini-4GL. ‘Transformation Menu’ is used to add behaviour to the node ‘Project_Editor’.

Figure 9.7 shows the implementation of the 'Transformation Menu'.

```
default transformation menu for concept Project_Component
  section(context actions tool) {
    submenu
      "Quantity";
      items
        action
          text (editorContext, node, model, pattern)->string {
            "Quantity--";
          }
          can execute <always>
          execute (editorContext, node, model, pattern)->void {
            node.quantity -= 1;
            editorContext.getEditorComponent().update();
          }
          icon <none>
          tooltip <none>
        ----------
        action
          text (editorContext, node, model, pattern)->string {
            "Quantity++";
          }
          can execute <always>
          execute (editorContext, node, model, pattern)->void {
            node.quantity = node.quantity + 1;
            editorContext.getEditorComponent().update();
          }
          icon <none>
          tooltip <none>
  }
```

Figure 9.7: Transformation Menu for Project_Component

The HTML output is shown in figure 9.8.

C:/Users/Majd/Documents/Tools/Gold%20Parser/Samples/mini4gl/mps/PharmacyX.html

PharmacyX

Code: #A100

Domestic

Description

ParmacyX is a 4GL programming language that illustrates the process of creating 4GL for Domain specific languages

Price: 1080$

components:

Name	Quantity	Treatment
Belbuca	1	treatment of moderate to severe chronic pain
Amoxicillin	1	treat many different types of infection caused by bacteria
Calan	1	treat heart rhythm problems

Figure 9.8: HTML Output for the New Mini-4GL Language

Creating a sample 4GL language allowed the construction of an abstract syntax tree for the given language grammar. The abstract syntax tree was traversed using an automated, that is program-controlled method, to generate output using the MPS language workbench.

9.3 Reverse Engineering the New Mini-4GL Language

This 4GL, unlike the languages in section 9.1, was a model-driven SQL-like 4GL (that is - the same class of 4GL as Uniface is), and so the translation as described in this section was a demonstration that the author's methodology worked on a 4GL of a class similar to Uniface. Demonstrations of the same methodology on other 4GLs of that class were not possible due to their unavailability, being of a proprietary nature.

Creation of the new Mini-4GL enabled writing code such as that as shown in figure 9.1. This in turn led to the ability to demonstrate the effectiveness of using the author's syntactical reverse engineering techniques on it, to demonstrate compatibility between the grammar rules of the derived output (in HTML) against a known input, being the Mini-4GL.

The code was parsed according to the language grammar rules, where the first grammar rule was defined as in figure 9.2.

The known types for source code editors were ‘Source Editors’, and ‘Projectional Editor’, Völter

and Lisson [338]. In the former the parser reads the input and parse it. The parser validates the source code and passes the generated AST across to the compiler for code generation. The compiler then generates its equivalent code in the target language (commonly machine code). Source code editing is illustrated in figure 9.9:

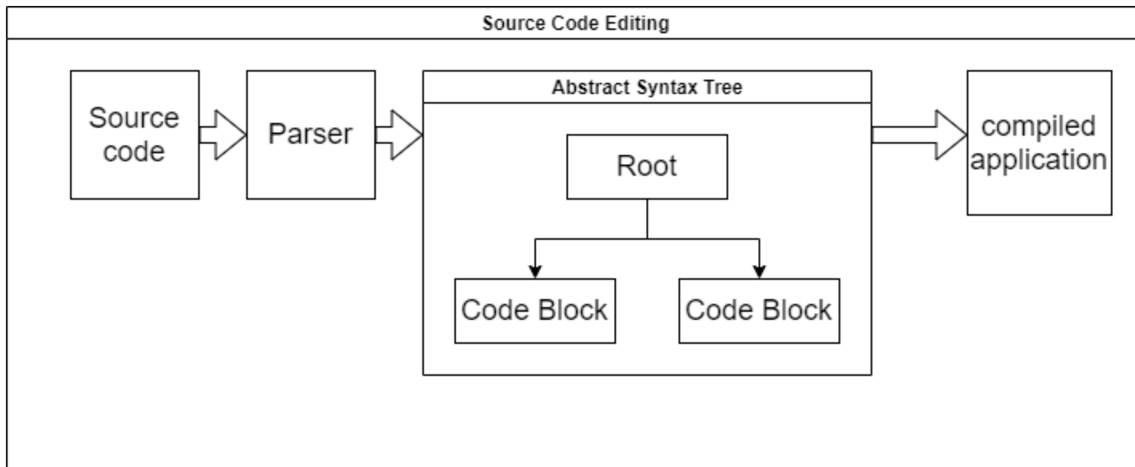

Figure 9.9: Source Editor Code Generation Process

The Mini-4GL editor was a ‘projectional editor’ where the editor recognised the relationships across the written lexemes. This meant that only valid syntax could be written in the editor.

The AST was auto-generated as the content was being edited. Figure 9.10 provides a schematic diagram of how such projectional editor's function.

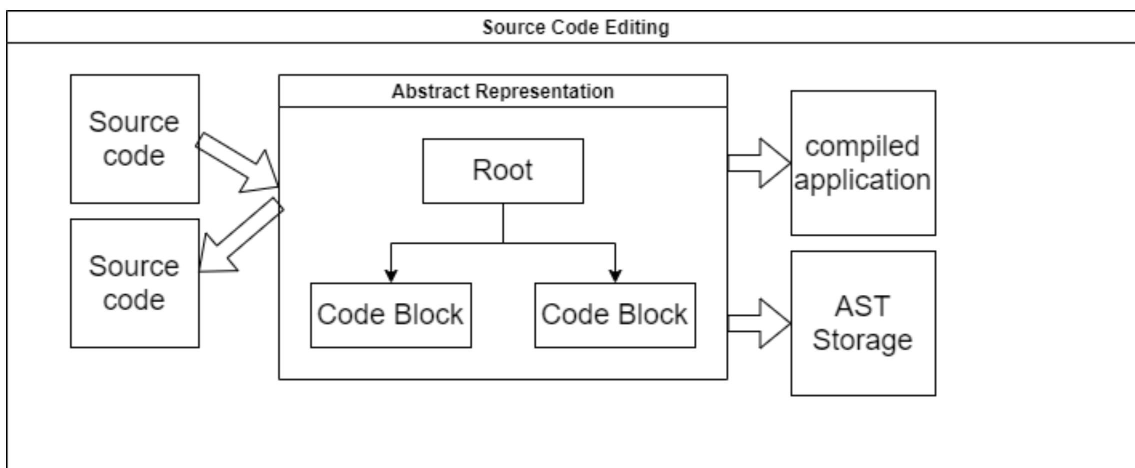

Figure 9.10: Projectional Editor Code Generation Process

The Mini-4GL was designed to take the following as input:

- Project name,
- Code, and
- A list of item's quantity, name, and price.

It has been designed to generate the following output:

- HTML file with valid HTML syntax,
- CSS style sheets attached to the HTML file, and
- Processed content from the input.

For this Mini-4GL, HTML and CSS grammar rules were redefined, according to each language's standard and utilised for the purpose of Mini-4GL code generation. Therefore, the Mini-4GL does not cover every possibility for coding in HTML and CSS but is just sufficient to deliver its purpose (an output of styled table of pharmaceuticals with a header and description). Figure 9.11 shows the template programming code generation snippet used to generate the output.

```
[root template
input Garage_Project]

html $[html file]

<!DOCTYPE html >
<html>
  <link rel="stylesheet" href="css/style.css"></link>
  <body>
    <h2>$[name]</h2>
    <h3>Code: $[#123]</h3>
    $IF$[<h3 name="" style="color:red;">Export</h3> ]/
    $ELSE$[<T <h3 name="" style="color:green;">Domestic</h3> T> ]
    <h3>Description</h3>
    $LOOP$[<p>$[text]</p> ]
    <h3>Price:  $[1000$]</h3>
    <h3>components:</h3>
    <table>

      <tr>
        <th>Name</th>
        <th>Quantity</th>
        <th>Material</th>
      </tr>
      $COPY_SRCL$[<row></row> ]
    </table>

  </body>
</html>
```

Figure 9.11: Template Programming to Generate PharmacyX Output

The transformation code in appendix H reflects the LALR parser method used to translate Uniface. The results of translation process shows exact match with figure 9.8. This accuracy of the reverse engineering of the Mini-4GL was due to:

- Known existing standards for both HTML and CSS,
- Having a complete understanding of how the Mini-4GL was designed and worked,
- The lack of complexity in design.

9.4 Conclusion

This chapter firstly covered (section 9.1) applying the methodology discussed in detail for Uniface in preceding chapters, on two other 4GLs (Informix and Apex), albeit on a much smaller scale.

It then turned to a very small original Mini-4GL, of limited scope, created by the author originally

for exploring how 4GLs work (section 9.2) and went on to cover (section 9.3) the successful reverse engineering of that.

Chapter 10 Results which follows, gives the results of the author's research. This is in the form of comparing the abstract syntax trees (in the form of chunking trees) generated from code in the source language (Uniface) against similar trees of the syntax generated from the output target language (in C#), for a number of different language syntax control structures.

This has also been provided for Uniface, and to a more limited extent for Informix, Apex and the Mini-4GL for each to the extent of the grammar rules which were covered.

Chapter 10

Results

This chapter provides results to show that the reverse engineering process set out in this thesis was successful. Success has been measured by the criteria that it produces valid code as shown in appendix A. This code compiles in the target language (C# in this case) and has been translated from the source 4GL language (in this case Uniface), with equivalent syntax. The syntax has been shown in chunking tree form, derived from the Abstract Syntax Trees (ASTs) for each language, with a range of different grammar rules for each.

Section 10.1 discusses the characteristics of programming (and natural) languages, in terms of syntax, semantics and standards. Abstract syntax trees can be used to describe a language's syntax, but are, in themselves, too complex to make comparisons possible.

Section 10.2 tackles this issue by firstly comparing various search methods. It then moves on to discuss features specific to ASTs. Specifically it discusses possible tree structures that could be used to simplify comparisons between the source (Uniface 4GL syntax) AST and the destination or target language AST (C#) for any particular language syntax.

Chunking trees were the chosen method of such simplification and section 10.3 (Using Chunking Trees to Compare ASTs for Source Language and Target Language Syntax) describes in detail the comparison for a given syntax structure as Run 1 of the Uniface research results.

Section 10.4 then gives summary results of such comparisons for a number of syntactical structures where the comparison was complete, partial or not possible due to the structure being a language specific syntactical feature.

Section 10.5 provides an evaluation of the results, as provided in the previous section.

10.1 Language Syntax and Semantics

Programming languages can be defined by their syntax and semantics leading to their specification. The syntax is the structure of alpha-numeric characters used as an input to make a program. The semantics are built into the language's compiler giving the syntax a behaviour. The specification is the combination of syntax and semantics as defined in the language's standard and/or reference manual.

Semantics gives a meaning to the syntax and alters a programs instruction's behaviour.

This research has been confined to developing a syntactical reverse engineering method, so the semantic or behavioural aspects of the reverse engineered output code has not been considered in its scope, or these results.

In the absence of any other meaningful metric, a comparison of the AST structures of the source code extracted and derived from the case study system (in this research Uniface) and that produced as transformed output (in this research C#) has been used as a useful starting basis for measuring the success or otherwise of the output produced.

Figure 10.1 represents the AST structure of a sample C# program given below. The tree structure in Figure 10.1 shows the complexity of even a very simple application, and this complexity grows as the application syntax expands. The tree structure in figure 10.1 is determined by the language rules (i.e. the number of branches and the names of nodes and leaves). Semantics define the tree execution path where different languages can have different execution paths even if they have similar ASTs.

```
1 class x {
2   function y() {
3     if (z == 10) {
4       var m = 1;
5     }
6   }
7 }
```

Figure 10.1: Abstract Syntax Tree for a Sample C# Application

The following is a pseudo-code example for a conditional statement.

```
1       if (x == 2 and y == 10) then
2           // if statement block
3       end-if
```

The 'If' block will execute if and only if both variable 'x' holds the numerical value '2' and 'y' holds the numerical value '10'. Syntactically, this condition is alike for every programming language that implements an 'If' condition. However, languages vary semantically when the condition is evaluated. Some languages evaluate both conditions for 'x' and 'y' and then apply the 'and' operator to determine the output Boolean value of the whole condition, which then determines the execution path, either into the 'If' block or not. Other languages evaluate the left side of 'and' operator and only evaluate the right side if the 'left' is true. This is because the 'right' becomes irrelevant if 'left' is false. Both implementation variants yield the same results, but the latter variant makes its programming language more optimised than the other in terms of compilation time (and will be so at run time also).

The following list describes the three different types of semantics according to Schmidt [339].

- **Operational:** where the program input determines the computational steps,
- **Denotational:** where the relation between input and output is determined mathematically,
- **Axiomatic:** where properties about input and output can logically describe the program.

Whether the three point of views of semantics described in the list above covered all situations, was also challenged by Schmidt [339]. For example, functional programming languages can pass functions as parameters as well as variables. Therefore, a formal execution path cannot be determined at compile time.

Inheritance in object-oriented languages also enables functions to pass parameters of different implementation variations of objects that inherit from one class. This forces the compiler to cast the parent object into the type passed into the function. Therefore, this research has not evaluated the semantics of either input or output languages but has treated them as black-boxes and just compared the AST of each.

10.2 Comparing Abstract Syntax Trees for Input and Target Languages

A comparison of ASTs of the input source programming language against the output produced in the target language was going to be one of the principal method of determining whether the author's program 'achieved success'. Therefore a suitable methodology for comparing trees (in this case ASTs) needed to be chosen.

The following sections (10.2.1 and 10.2.2) discuss tree search methods for language syntax comparisons.

10.2.1 Tree Search Comparison Methods

Comparing languages syntax is commonly used to detect code changes and detect clones and plagiarism in code. Code comparison ignores if possible names for both function and variables, and line swaps. Comparison is done using the following three methods, according to Cui et al. [99]:

- **Text-based search:**
  This type of comparison is agnostic to the language rules and compares the text as characters, words, sentences and paragraphs. This method looks for exact matches and is unable to compare code that contains changes in name or order of statement.
- **Token-based search**:
  This uses slightly more advanced techniques than former, where the engine creates tokens for a series of characters and aims to compare tokens. This technique can determine changes in names for functions and variables but is unable to detect line swaps. Tokens were discussed previously in section 8.3.4.
- **Syntax-structure search:**
  This method uses the language AST to detect all above types of changes and is considered a more advanced technique than other two.

All the three search methods above have been used to detect the changes, clones and plagiarism in the same subject language Cui et al. [99], Torres et al. [340] and Neamtiu et al. [101].

Instead of using syntactical search for code change detection, the research in this thesis has used the technique to represent the language's syntax. It also offers richer capabilities to traverse the language's control structures, as well as derivations that can be extracted from it as discussed in the following section.

10.2.2 Abstract Syntax Trees and their Derivations

As discussed in section 3.4.3, ASTs are generated from the source code of a programming language and represent in a formal form the syntactical structure of the code. It follows therefore, that if the reverse engineering process produces generated output code with a matching AST, then it will be syntactically valid in the output language. The concept of being syntactically valid, in turn determines that the code will compile and execute. The semantical behaviour of the code is beyond the scope of this thesis.

An AST has structural features that define its particular tree structure. Zhang et al. [341] define the following AST features of how a particular set of nodes being considered, might be positioned:

Minimum Complete Tree (MCT):
A MCT consists of the nodes in consideration all stem from a common parent, which is considered the new root of the MCT.

Path-enclosed Tree (PT):
A PT is formed of two common sub-trees which contain the nodes being considered, including other nodes that lead to a common root.

Context-sensitive Path Tree (CPT):
A CPT is a path-enclosed tree that is built of the original nodes under consideration plus the nodes immediately to the right and immediately to the left, in order to form a new tree, including the original nodes plus the new nodes.

Chunking Tree (CT):
A CT is a path-enclosed tree where the nodes are direct descendants of the root node - that is without any nodes that build intermediate structure. Therefore a CT contains the nodes being considered, their parent nodes, and a common root.

Context-sensitive Chunking Tree (CCT):
A CCT is a CT tree extended using the same method used in CPT to include the left-most and right-most nodes to the first and second nodes being considered.

Figures 10.2, 10.3, and 10.4, show MCT, PT, and CT applied to the AST of the following Uniface snippet:

```
1    entry LGET_FROM_ADDRESS
2    if ($$USERNAME = "")
3    $$USERNAME = "hermione"
4    endif
5    TFROM_TEXT = "%%$$USERNAME%%%@hogwarts.ac.uk"
6    end
```

Assuming that the nodes under consideration are '$$USERNAME' and the empty string in the condition line 2 of the snippet above, then the MCT is:

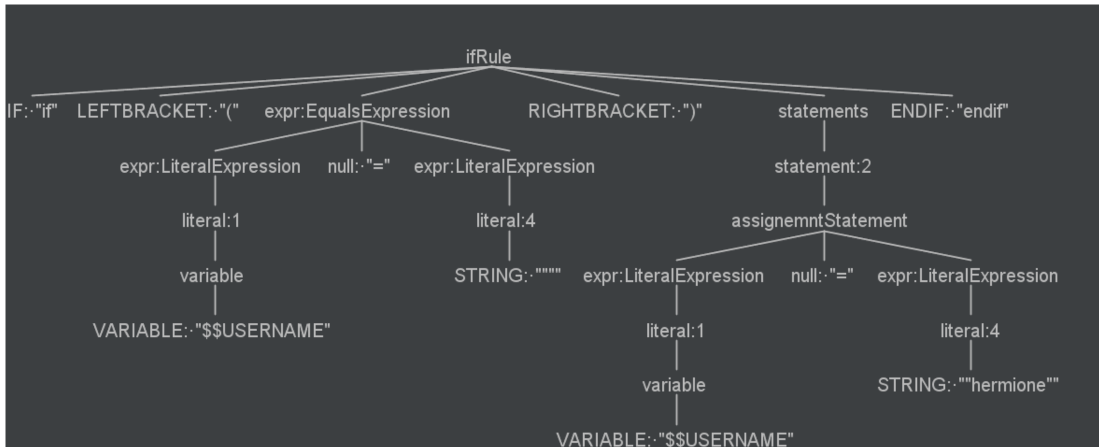

Figure 10.2: Minimum Complete Tree (MCT)

A PT for the same two nodes is illustrated in figure 10.3

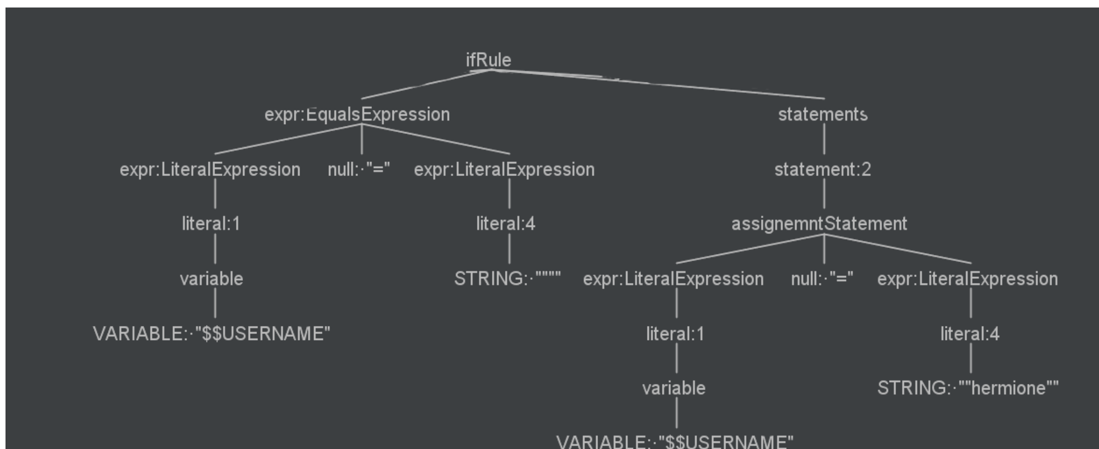

Figure 10.3: Path-enclosed Tree (PT)

A Chunking Tree for the same two nodes, is shown in figure 10.4

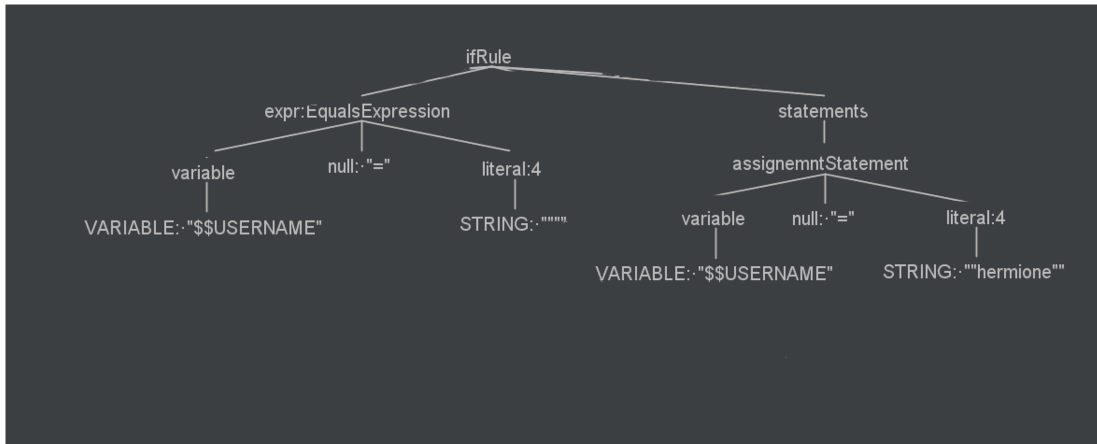

Figure 10.4: Chunking Tree (CT)

10.3 Using Chunking Trees to Compare ASTs for Source Language and Target Language Syntax

Chunking trees were chosen as the preferred method of comparing ASTs between two different programming languages for this research. This was because they refine the tree to include only the nodes that hold tokens (language grammar rule labels) and simplifying their comparison. Other forms of trees also include unwanted intermediary nodes, which just add noise.

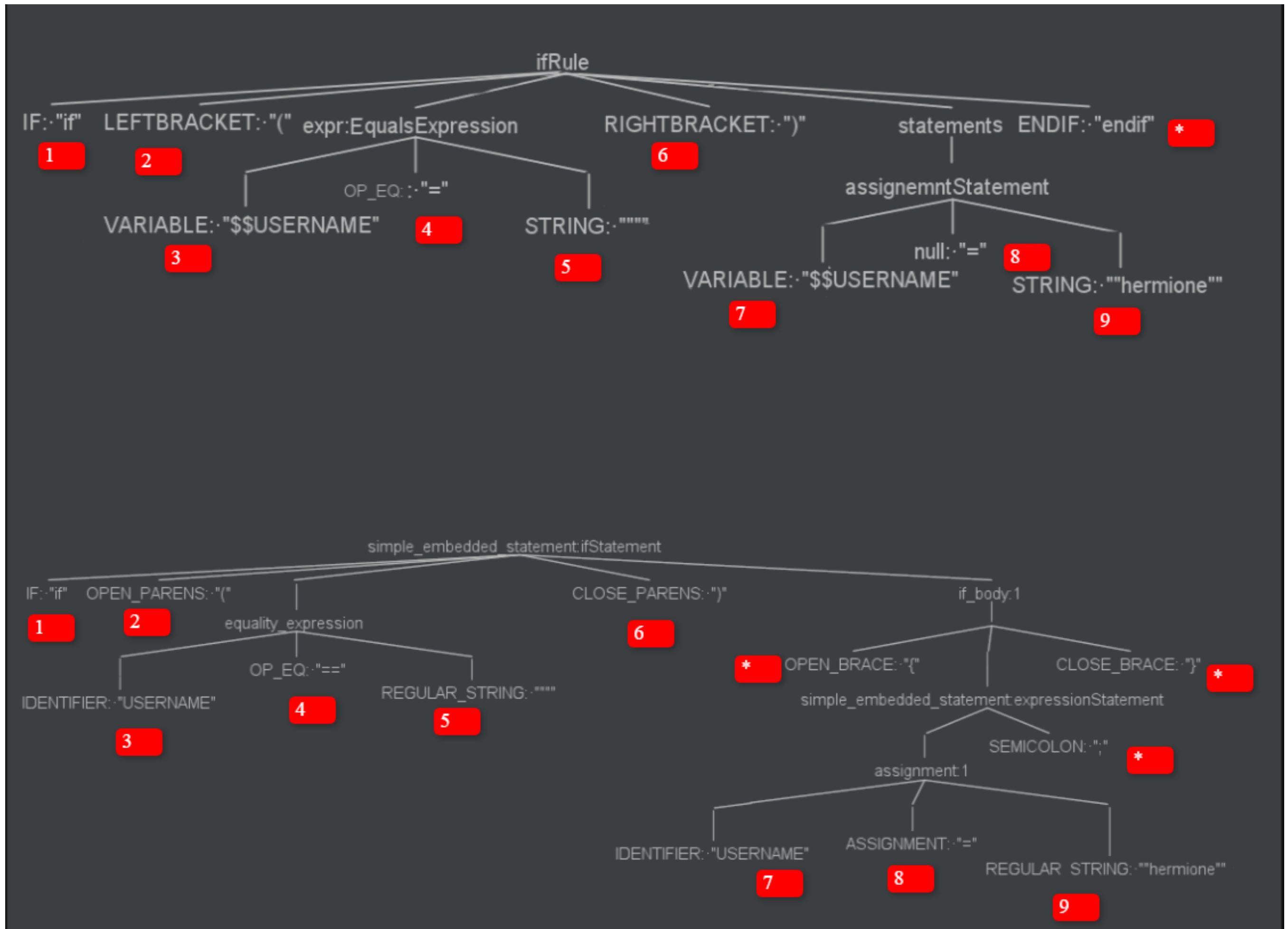

Figure 10.5: Run U1: Chunking Trees to Compare Uniface (Tree A - top) to C# (Tree B - bottom) Chunking Trees

To test the author's system ASTs chunking trees were compared for the source programming language (Uniface) against the output target (C#) for various aspects of language syntax to determine whether and to what extent they matched.

An initial trial (Run U1) used the full example code snippet above, previously used to demonstrate different tree comparison types. This run is intended to exercise syntax where the execution falls in the If-block and the variable within is loaded with the string value ‘hermione’.

The result provided

- valid destination code (C#). This can be found in appendix A (Output Code in C# from Input Code in Uniface), and
- correct results i.e. matching chunking trees derived from each code's ASTs, which demon-

strated a complete and syntactically correct translation (figure 10.5).

Figure 10.5 shows the chunking tree for both languages Uniface (top), and C# (bottom) which match exactly with respect to each language's syntax.

Nodes numbered 1 to 9 represent the language syntax grammar rules using the format:

grammar rule name + ':' + exact token value as seen in the code.

Non matching language-specific nodes are marked using the asterisk symbol '*'.

Table 10.1 below provides explanations of figure 10.5 for

- each matching node (numeric labels) and
- non-matching node (* label).

Where descriptions differ for each programming's chunking tree, language A refers to the source language (Uniface) CT and B refers to the target language (C#) CT.

Node	Node Type	Node Content	Description
1	Control statement	If	If keyword - used in both languages to denote the start of ‘If’ block
2	Symbol	Left bracket ‘(’	Conditions in both languages are grouped in brackets
3A	Identifier	‘$$USERNAME’	Uniface variable naming allows for the ‘$’ symbol
3B	Identifier	‘USERNAME’	The dollar symbols are omitted in C# where the symbol is prohibited in variable names
4A	Boolean Comparison	‘=’	Boolean comparison for Uniface
4B	Boolean Comparison	‘==’	Boolean comparison for C#
5	String	""	Empty string for both languages
6	Symbol	Right bracket ‘)’	Closing bracket for the Boolean condition
7A	Variable	$$USERNAME	variable $$USERNAME for Uniface
7B	Variable	USERNAME	variable USERNAME for C#
8	Assignment	‘=’	Is the same for both languages. used to assign values to variables
9	String	‘hermione’	A string value assigned to the variable
*A	End control	’endif’	Uniface specific to terminate the If-block statements
*B	Symbol	left curly bracket ‘{’	Is a C# specific and denote the start of If-Block
*B	Symbol	right curly bracket ‘}’	Is a C# specific and denotes the end of If-Block
*B	End of statement	‘;’	The semicolon is used to terminate statements in C#

Table 10.1: Run U1: Chunking Tree Comparison Uniface to C# - Explanation of Labels

Complete translation

The above (Run U1) is an example of a ‘complete translation’ where a complete translation has been defined as where the target language syntax was valid and intact. In addition, the target syntax did not result in any information loss and it was able to be parsed using a third-party (non-biased) parser tools.

Other types of translation encountered during other tests were:

- ‘partial complete’,
- ‘failure’, and
- ‘grammar not available’.

Each is explained below.

Partial complete translation
This was when the translation is done successfully but parts of the code were missing.

In the case of Uniface it could be due to missing elements which Uniface attempted to retrieved from its meta-data database. For example, Uniface used internal variables called ‘status’ to determine the outcome of the most recent transaction on the database. Uniface managed this variable and allowed developers to use it. The translator generated the variable in the target languages (C#) but its value was constant. This could lead to a syntactical valid result which was the main concern of this study. The missing content could be manually identified and fetched into the translator to append the generated code with routines that replaces the ones fabricated into Uniface compiler and not publicised.

Failure
This was where translation is noticed to fail for external commands and missing information. Uniface allows for external calls to external applications. In this case-study it was found that this occurred when external calls were made to run programs to deliver functionality that Uniface is incapable of implementing. For example, merging document's content where the first source of information came from email and second source was a Microsoft Word template.

The following example snippet of Uniface code demonstrates code which would fail to translate:

```
1      CMDLINE = "x:\utilities\blat.exe %%"c:\uniface\messages.txt%%"
2      -server %%"cencored.essex.ac.uk%%"
3      -subject %%"%%$TEMAIL_SUBJECT_TEXT$%%%%%" -f %%"%%FROMADDRESS%%%%%"
4      -to %%"%%TOADDRESS%%%%%" -q"
5     ;askmess/error "CMDLINE %%CMDLINE%%%"
```

In this example, ‘blat.exe’ was inaccessible and its behaviour was unknown. Therefore, a translation mechanism could not be provided. In addition, parameters such as ‘server’, ‘subject’, ‘f’, ‘to’ and ‘q’ were obscured. Developers who built the case-study used in this research have also, utilised Python language to build routines to deliver complicated functionality that was believed to be infeasible in Uniface. Translation of such routines are currently seen as an intractable problem.

Grammar not available
In such causes the translation fails due to the absence of grammar rules. The author created Uniface grammar rules which covered the following code blocks:

- controlling blocks such as (‘If’, ‘when’, and ‘switch cases’),
- loops (‘for’, and ‘while’),
- functions declaration, and
- built-in functions ‘retrieve’, and ‘sort’.

Uniface has more built-in functions where its grammar rules could be written and implemented using the same methods that this research proposes. This research offered a proof of concept which could be extended for the benefit of approaching Uniface translation from engineering/commercial point of view. This could require a large budget and a team dedicated for writing grammar rules.

10.4 Summary of Results: Comparison Runs

The results are first presented in detail for Uniface, followed by the other 4GLs Informix, Apex and Mini-4GL.

10.4.1 Summary of Results - Uniface Comparison Runs

Table 10.2 below gives summary results for comparisons of the ASTs in chunking tree form, for the Uniface input and C# outputs for all the grammar rules as implemented.

Run U1 is the summary for the scenario explained in detail in figure 10.5 and Table 10.1 above. Detailed explanations have not been provided for each run, as their outcomes were either analogous to Run U1 (compete translation) or differ for the reasons explained above.

This table summarised the attempts to translate a variety of ESIS syntax code blocks (code types) grouped by their logical control statements. Examples are ‘If’ conditions, nested ‘If’ conditions, loops triggered by the ‘while’ keyword, comments and external calls.

Table 10.2 presents the results as follows:

- Column 1: ‘Run ID’ - A Unique identifier (U in this case standing for Uniface),
- Column 2: ‘Code Type’ - The generic syntax distinct feature identified,
- Column 3: ‘Description’ - Explanation of the syntax under consideration,
- Column 4 ‘Result’ - The success level of translation for that syntax.

Each run in the table U1 to U10 is a translation run for the code type, labelled in column 2 (Code Type). Therefore, each run in this table can be expanded to a table analogous to table 10.1.

Run ID	Code Type (distinct feature)	Description	Result
U1	Control statement (‘If’ condition)	If statement and value assignment to a variable *(explained in Table 10.1)*	Complete translation
U2	Loop (‘while’)	Iteration (if the condition is true) over associative list and assign values to variables	Complete translation.
U3	Switch case	The code flows through a selected statement stipulated by a variable value	Complete translation
U4	Variables definition & calls to methods within same module	This block of code defines variables and passes them into functions defined in the same module	Complete translation
U5	Control statement	Example of nested ‘If’ and ‘else’ statements	Complete translation
U6	External call	The code contains `spawn "CMD /C Copy n:\word\stu400m* c:\uniface"` which contains ‘spawn’ command which is unknown to the user and calls external content that was not made available to the author	Failure
U7	List of comments	Uniface uses semicolon to comment out a line and translates to // in C#	Complete translation
U8	Loop and control statement	Nested If statements within a ‘while’ loop	Complete translation
U9	Message dialogue	Display string content in a message box form	Complete translation
U10	Unknown grammar	A call to statement ‘forentity’ `forentity "AMS_READER_ROOM"` for which the author has not created a grammar rule to cover this type of statement	Grammar not available

Table 10.2: Summary Results Runs U1 - U10: Chunking Tree Comparisons Uniface to C#

10.4.2 Summary of Results - Informix 4GL Comparison Runs

Informix is syntactically very different from C#. So there may not be a 1 to 1 correspondence between the code in one language and the code in the other.

The generated syntax tree for Informix was too large to fit on a page, so has been split to simplify comparisons.

The studied function is displayed below

```
1     MAIN
2        DEFINE i SMALLINT
3        FOR i = 1 TO 10
4            DISPLAY i
5        END FOR
6    END MAIN
```

Where 'MAIN' defines the function heading. Then variable 'i' is defined with 'SMALLINT' data-type where its value ranges from $-2^{15}(-32,768)$ to $2^{15-1}(32,767)$. The equivalent data-type in C# is 'Int16'.

The following chunking tree (figure 10.6) shows the syntax for the first part of the code (lines 1, 2, 6).

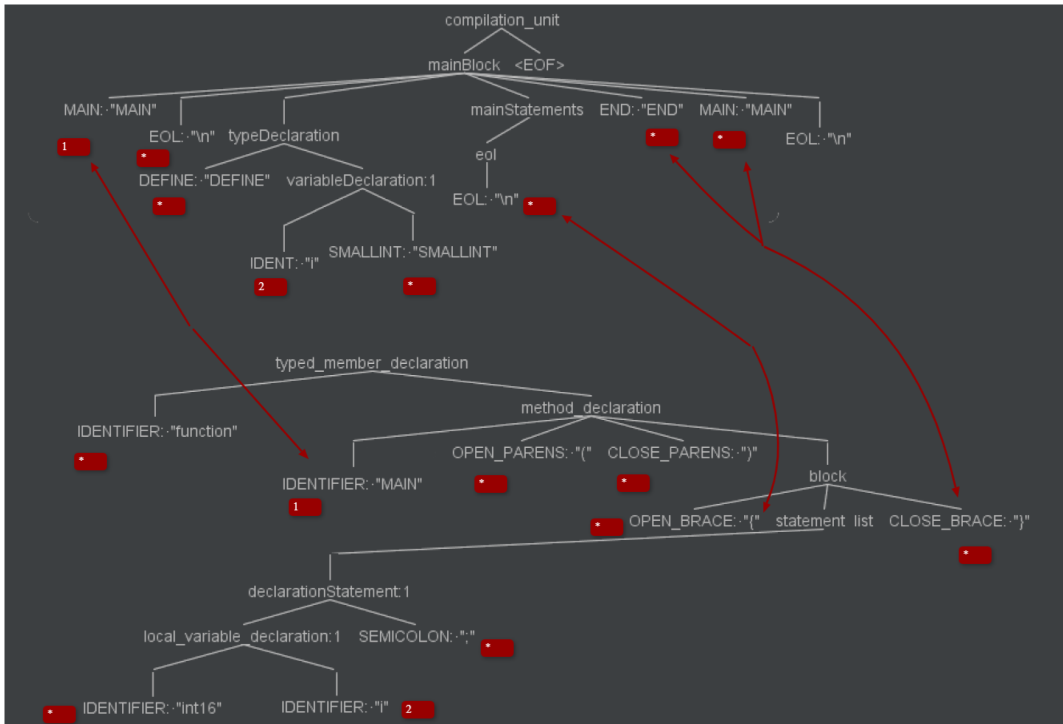

Figure 10.6: Informix Function Definition (top) Chunking Tree - Compared to C# (bottom)

Table 10.3 displays the results discussion for the comparison in figure 10.6.

Run ID	Code Type (distinct feature)	Description	Result
I1	'MAIN'	Function name	Complete translation
I2	Variable definition (aka. identifier in Informix)	Variable declaration	Complete translation.
I*	'Define'	Informix specific to declare a variable	Complete translation
I*	'SMALLINT'	Informix Data-type of range from $-2^{15}(-32,768)$ to $2^{15-1}(32,767)$ - translates to 'int16'	Complete translation
I*	'END MAIN'	Informix function terminator - translates to opening and closing curly brackets in C#	Complete translation

Table 10.3: Summary Results Runs I1 - I2: Chunking Tree Comparisons Informix to C#

The notation ‘I*’ refers to language specific syntax which can be translated but the syntax is not expected to be one of the following:

- Exact match if the target language uses the same syntax,
- Target language equivalent syntax,
- Omitted due to the target language specification.

The inner for-loop (lines 3, 4, 5) for Informix is shown in two forms. Firstly, it has been defined using railroad syntax, according to Informix [13], as in figure 10.7. Secondly, figure 10.8 shows a comparison using chunking tree for the legacy language Informix and the translated output in C#.

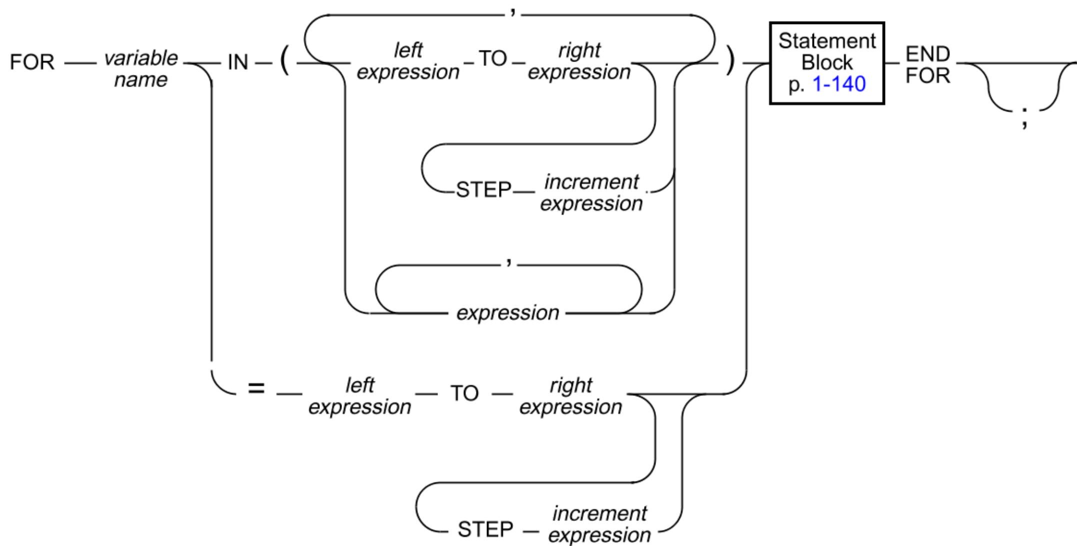

Figure 10.7: Informix For-Loop Railroad [13]

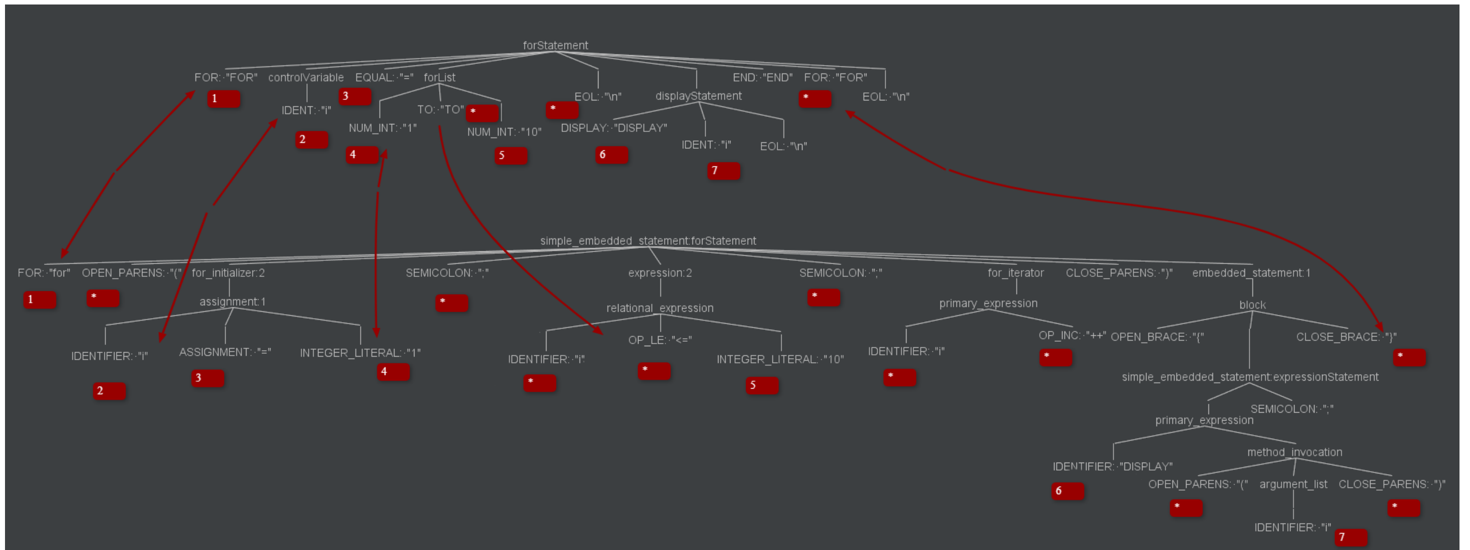

Figure 10.8: Informix Inner For-Loop (top) Chunking Tree - Compared to C# (bottom)

Table 10.4 below gives comparative results for the runs covering the Informix for-loop as shown in the chunking tree (figure 10.8), and as a railroad diagram (figure 10.7) above.

Run ID	Code Type (distinct feature)	Description	Result
I1	‘FOR’	loop starting point name - translates to ‘for’ in C#	Complete translation
I2	identifier ‘i’	Loop variable declaration	Complete translation.
I3	‘=’	Loop variable initial value	Complete translation
I4	‘1’	Integer value for variable ‘i’ when the loop starts	Complete translation
I5	‘10’	Loop top end.	Complete translation
I6	‘DISPLAY’	Function call - translates to ‘DISPLAY(...)’ in C#	Complete translation
I7	variable ‘i’	Using the variable in the loop block	Complete translation
I*	‘END FOR’	Terminates the loop - translates to opening and closing curly brackets ‘’, ‘’ in C#	Complete translation

Table 10.4: Summary Results Runs I1 - I7: Chunking Tree Comparisons Informix to C#

The above results show that the author's method produced complete and matching syntactical translations for Informix, for the parts of that language that were studied.

10.4.3 Summary of Results - Apex Comparison Runs

The translated text for Apex is as follows

```
1   @IsTest
2   private class TestRunAs {
3   public static testMethod void testRunAs() {
4       String debugginValue = 'Runtime started OK';
5       System.debug('Runtime status:' + myString);
6       }
7   }
```

The relevant AST or chunking tree was too large to fit on the page. Therefore, it was segmented and studied in segments, in a similar manner to that for Informix above.

The first part includes the attribute shown at line 1 and a comparison of the chunking trees for the legacy language AST and the output in C# is illustrated in appendix E figure E.3.

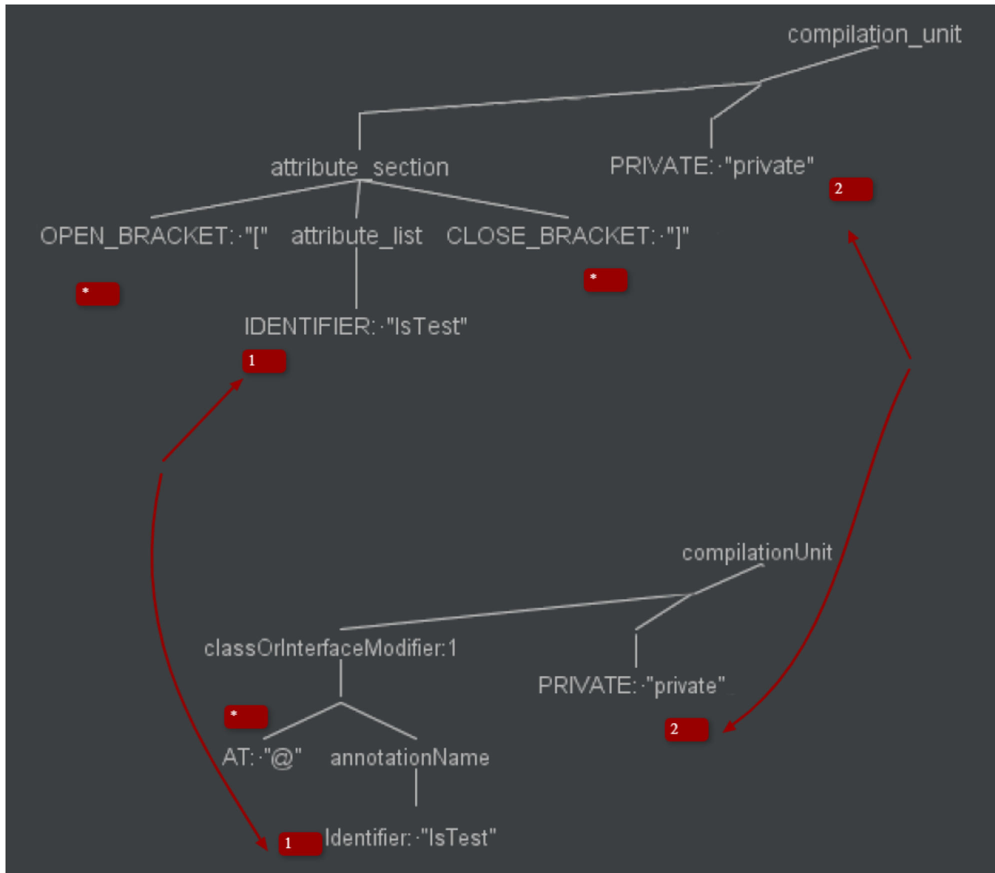

Figure 10.9: Apex Attribute (top) Chunking Tree - Compared to C# (bottom)

The second AST comparison was for the class definition (code line 2) and function definition (code line 3) as shown in figure 10.10 below, with the results tabulated in table 10.5.

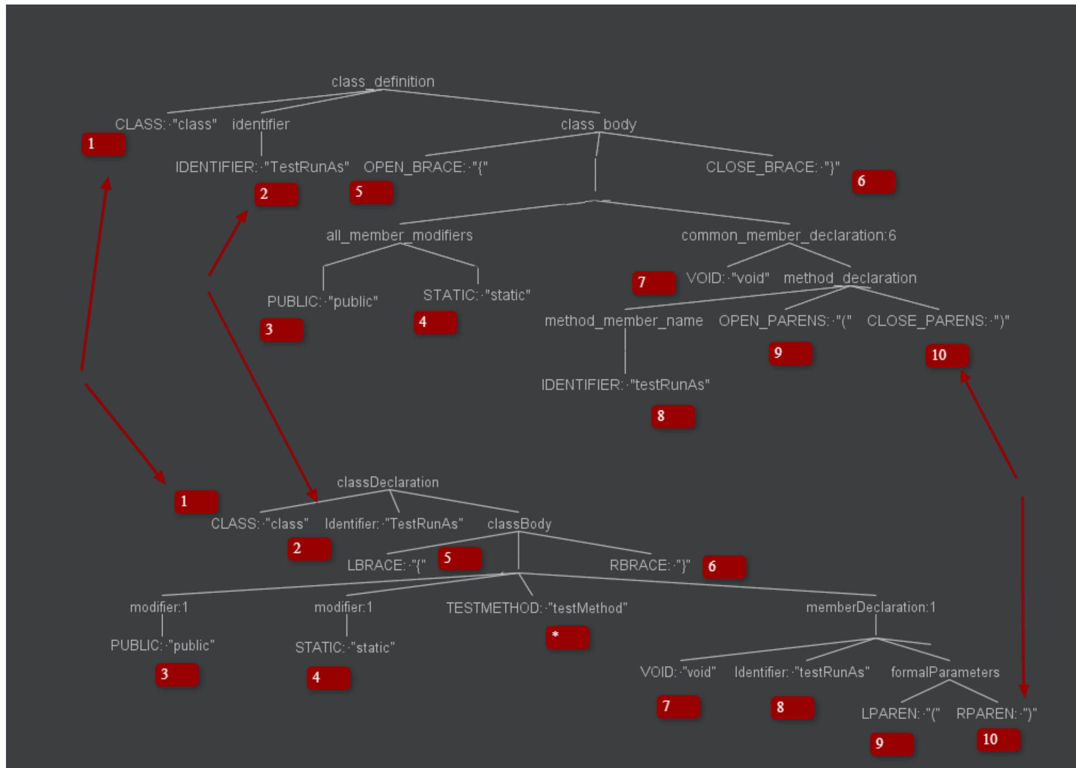

Figure 10.10: Apex Class and Function Definition (top) Chunking Tree - Compared to C# (bottom)

Run ID	Code Type (distinct feature)	Description	Result
A1	‘Class’	Class keyword that defines a new class	Complete translation
A2	identifier	Defines the class name	Complete translation.
A3	‘public’	keyword defines the modifier access level	Complete translation.
A4	‘static’	The keyword ‘static’ is used for memory management where the same instance of object is statically loaded into memory for all instances of the same type	Complete translation
A5	‘{’	Open bracket applied to both languages to start the class body block	Complete translation
A6	‘}’	Close bracket applied to both languages to terminate the class body block	Complete translation
A7	Return type	The return type for both languages	Complete translation
A8	Identifier	the function name	Complete translation
A9	‘(’	Opening bracket for arguments	Complete translation
A10	‘)’	Closing bracket for arguments	Complete translation
A*	‘testMethod’	Apex specific keyword found with test methods definitions	Complete translation

Table 10.5: Summary Results Runs A1 - A10: Chunking Tree Comparisons - Classes and Functions - Apex to HTML

The notation ‘A*’ refers to language specific syntax which can be translated but the syntax is not expected to be one of the following:

- Exact match if the target language uses the same syntax,
- Target language equivalent syntax,
- Omitted due to the target language specification.

The third part of the chunking tree displays the first part of the method definition (code lines 4 and 5). The results are shown as chunking trees comparing Apex legacy code with C# output code syntax (figure 10.11) and then these are described further in tabular form (table 10.6).

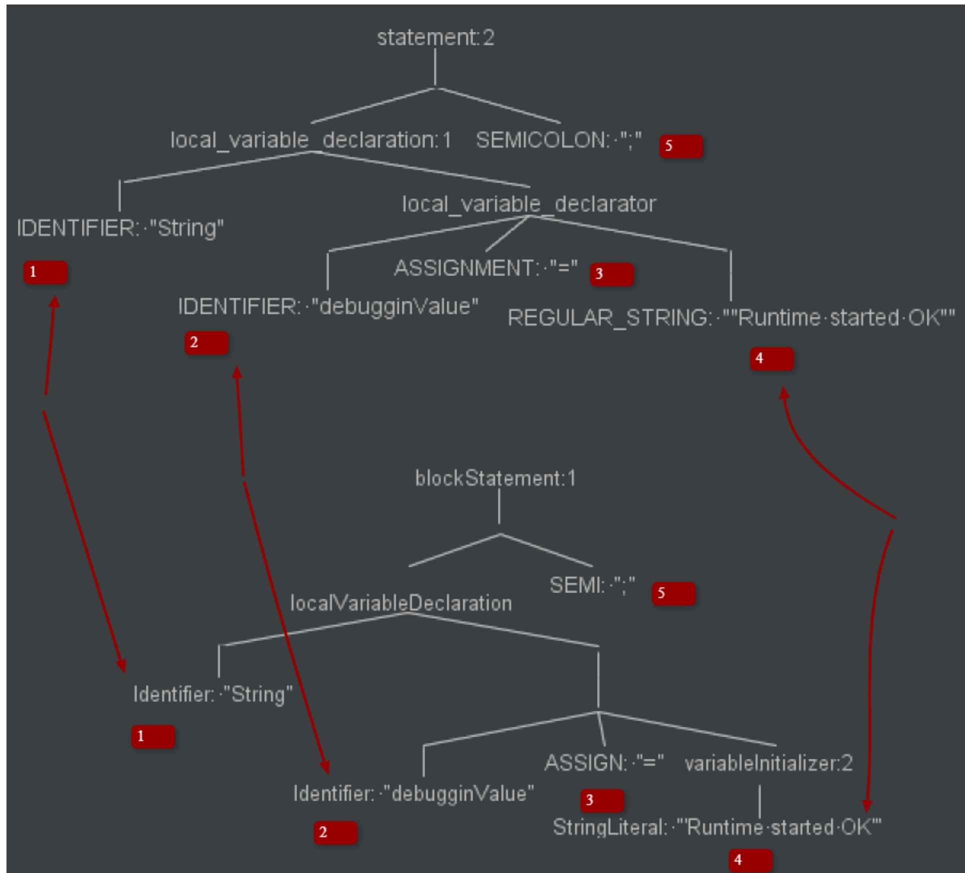

Figure 10.11: Apex Attribute (top) Chunking Tree - Compared to C# (bottom)

Run ID	Code Type (distinct feature)	Description	Result
A1	‘String’	Defines a new variable of the type ‘string’	Complete translation
A2	identifier	Defines the variable name	Complete translation.
A3	‘=’	Equal sign to assign a value into the variable	Complete translation
A4	fixed string	fixed value	Complete translation

Table 10.6: Summary Results Runs A1 - A4: Chunking Tree Comparisons - Method Definition Part 1 - Apex to C#

The fourth part of the chunking tree displays the second part of the method definition. The results are shown as chunking trees comparing Apex as legacy language syntax with C# as output language syntax (figure 10.12). These are then described further in tabular form (table 10.7).

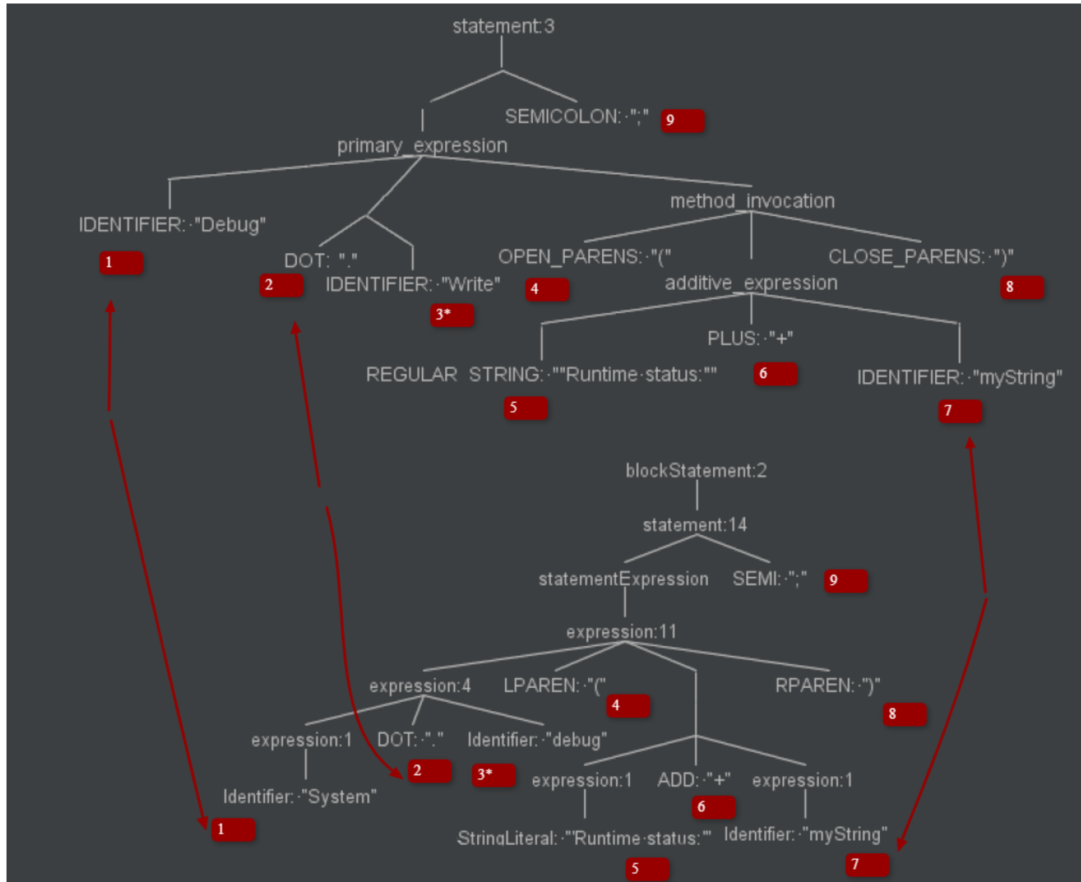

Figure 10.12: Apex Attribute (top) Chunking Tree - Compared to C# (bottom)

Run ID	Code Type (distinct feature)	Description	Result
A1	Library call	A call to the input/output (I/O) library where the print to console is defined ('Debug' in Apex and 'System' in C#)	Complete translation
A2	'.'	Dot - represents a call to method members	Complete translation.
A3	Write to console function	the function in the built-in library to display characters on screen	Complete translation
A4	'('	Open bracket for arguments	Complete translation
A5	Static string	A static value	Complete translation
A6	'+'	Plus sign do string concatenation in both languages	Complete translation
A7	Identifier	summon the identifier's value	Complete translation
A8	')'	Close bracket for arguments	Complete translation

Table 10.7: Summary Results Runs A1 - A8: Chunking Tree Comparisons - Method Definition Part 2 - Apex to C#

The above results show that the author's method produced complete and matching syntactical translations for Apex, for the parts of that language that were studied.

10.4.4 Summary of Results - Mini-4GL Comparison Runs

The chunking tree for Mini-4GL was too large to fit display on a page. Thus it was studied by segments. Due to the fact that Mini-4GL was designed to output styled HTML pages with CSS, it was also decided to reverse engineer it's grammar rules and set the output to be styled HTML with CSS. It was shown in figure 9.8 that both the generated HTML and the HTML resulted from reverse engineering it are a match in terms of the page's appearance. A syntactical comparison is depicted for run M4 in figure 10.13 and 'Summary of Results' commentary has been provided as table 10.8.

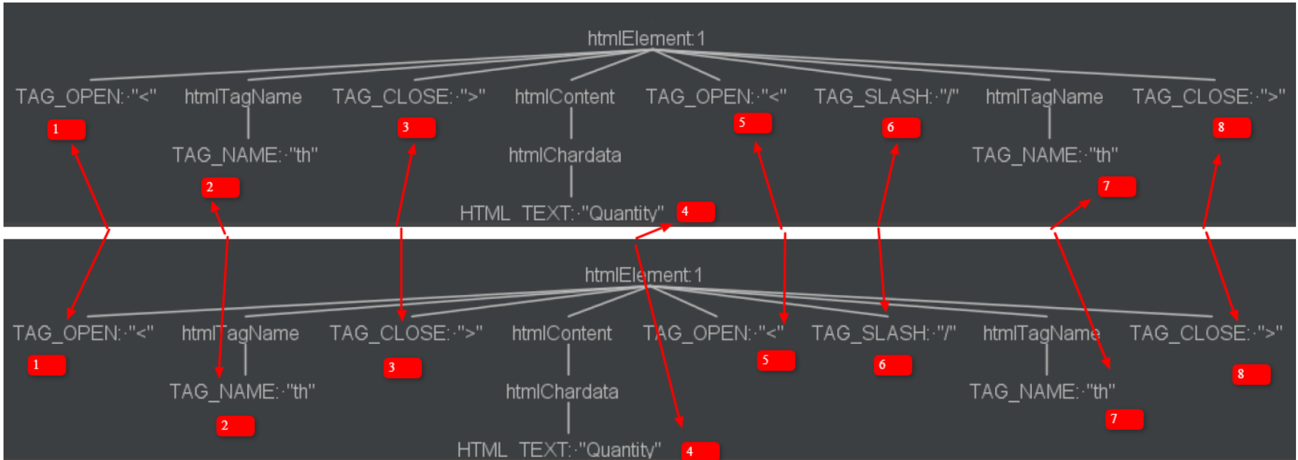

Figure 10.13: Run M4 'th' element: Chunking Trees to Compare Mini-4GL (Tree A - top) to HTML (Tree B - bottom) Chunking Trees

Run ID	Code Type (distinct feature)	Description	Result
M1	Tags opening and closing	Elements such as HTML, style, and body	Complete translation
M2	Custom CSS	Include statement for custom CSS file such as `<link rel="stylesheet" href="..."/>`	Complete translation.
M3	Tags used to style text	Tags such as 'h2', 'h3' and paragraphs 'p'	Complete translation
M4	Tables	Tables including the table opening and closing tag, table headers 'th', rows 'tr' and columns 'td'	Complete translation
M5	Inline styling	Custom CSS applied directly to an HTML element bypassing the custom CSS included	Complete translation

Table 10.8: Summary Results Runs M1 - M5: Chunking Tree Comparisons Mini-4GL to HTML

10.5 Evaluation of Results

This research aimed to transpile the syntax of a 4GL modular language (Uniface as an exemplar) into a 3GL. The results show that transforming the syntax as a preliminary step is achievable. This research proves that it was possible to translate the syntax into the target language (C#), for which the specification has been fully published by its author Microsoft [342].

The target code was syntactically valid and matching, which was the main goal of this research.

It compiled and executed using standard parsing tools - both those that were originally created by C# provider and also those using third-party tools (for example, C# grammar for ANTLR).

This research has applied a novel approach (chunking trees) to compare the similarities or differences in the syntax of the two different programming languages. Comparing chunking trees is a promising approach since chunking trees represent the source and target syntax in a unified form which can be used to asses the results syntactically. For complete translation result, the chunking tree would be a complete tree carrying all the expected nodes from the AST origin. For a partially complete result, the chunking tree just would plot the translated piece of code and omit the untranslatable code. Where a failure result was reported, it was impossible to establish a chunking tree.

The method demonstrated in this research was applied on various structures of code as shown in tables 10.2 and 10.8. Comparing the source chunking tree to the destination one has been a manual process that required expertise in the syntax of both languages, to determine whether the grammar rules are correctly matching or not.

A successful translation of one grammar rule syntax structure leads to successful translating for this syntax anywhere it was presented using the exact same form regardless of changes in its variable's names or values. This was due to the formal nature of programming languages.

This research did not preform analysis on the percentage it covered from the original case-study, due to the fact that the target has been approached using grammar rules and syntax tree structures, rather than on a line by line basis. Since the source language was proprietary with no published specification, and the research has been performed on a single case study, it has been impossible to measure what a 100% translation might be. An additional factor has been that some of the source code is not Uniface (e.g. Python), which the original case study developers included for additional functionality.

Further semantics study and data analysis are proposed in chapter 11 section 11.6 (Future Work).

10.6 Measuring Transformation Scope

Measuring the transformation scope (i.e what proportion of the language has been covered by the process), for Uniface or any other proprietary 4GL is based on the availability of a complete and accurate syntax - which by their nature would not have been published. In the case of the case study considered here, it cannot be guaranteed that these rules would be necessarily fully covered, even in complex systems such as ESIS. The ESIS system was built using Uniface version 9. ESIS may or may not covered all of the possible syntactical rules which that version of Uniface offers. Therefore, less accurate reverse engineering transformation results would be expected for Uniface.

Therefore the extent of such loss of accuracy would be unable to be determined without a similar exercise based on a number of other large Uniface based systems being similarly reverse engineered,

and even then the data would still be an approximation.

10.7 Conclusion

This research has been concerned with reverse engineering the syntax of a proprietary 4GL source language (in this case Uniface) into valid identical syntax in a destination or target language (in this case C#). The behaviour or semantics of the language was outside the scope of this research (section 10.1).

A meaningful results metric to compare the 'success' or otherwise of the author's method was to compare the input code syntax (in this case Uniface) against the output code syntax (in this case C#). This was done by comparing their abstract syntax trees (section 10.2). It was found that ASTs themselves were too complex and simplification was provided by using chunking trees for the comparison (section 10.3).

A detailed explanation of one particular Uniface syntax structure was provided for Run U1 in the same section.

Summary results based on a similar analysis were then provided (additional runs) for a number of other syntactical structures (section 10.4 and in particular for Uniface table 10.2) and the Mini-4GL in table 10.8. This provided an indication of whether complete or partial matching trees and therefore translation was possible for a range of syntactical structures, including language specific syntax.

In some cases no such translation was possible because the grammar rule had not been implemented by the author (this only being a proof of concept rather than a full implementation)

The following and final chapter gives an appraisal of the novelty provided by the thesis and a critical appraisal of the achievements of the research. It also looks ahead at what avenues might be available to build on the proof of concept code using the methodology described and future possible research avenues.

Chapter 11

Conclusions

This chapter starts with a brief overview of the main themes of the research in this thesis (section 11.1).

It then goes on, in (section 11.2), to consider whether and to what extent the Research Questions posed in section 1.7 have been answered.

Section 11.3 summarises the key contributions to knowledge this thesis has provided.

Section 11.4 covers areas related to improvements that could be made to the system and what might be done to complete it, in terms of:

- Providing coverage of a more complete set of grammar rules for Uniface (section 11.4.1),
- Automating the parts that could be readily automated (section 11.4.2).

Lessons learned in carrying out this research has been covered in section 11.5.

How other researches might continue and expand upon themes covered by the research in this thesis is covered by section 11.6 (Future Work).

The author has provided some concluding thoughts as section 11.7.

11.1 Recapitulation of Reverse Engineering, ESIS Case Study and Research

The software engineering field of reverse engineering programming languages mainly aims to achieve some or all of:

- increasing the understanding of programs written in that language,
- recovering the program's design, and

- translating from one language's syntax to another.

This thesis has focused primarily on translating Uniface syntax by parsing its grammar rules. The author of this thesis is not aware of any similar work - successful or otherwise.

This research has largely focused on one specific example 4GL (Uniface) and its implementation in one specific system (the University of Essex's ESIS system), as described in section 1.5.2.1. The University of Essex ESIS case-study was explained in section 1.5.3. The case-study consisted of Uniface meta-database, the student database, and the exported XML-like code within which the ESIS Uniface code was embedded. Recovering the design directly from the meta-database was not possible due to the limitation in recovering the structure and content of its tables, as explained in section 3.1.1.

This research paid particular attention to Uniface and other XML-like model-driven 4GLs. The research explored the potential of discovering a novel method, if possible capable of automation, to translate the syntax of a source code of an application written in Uniface for example, into another language (C# for example).

Uniface as an example of a 4GL and a model-driven language 1.5.3.1 was compared to a modern language '.NET' (see sections 1.3.3 and 1.3.4).

This thesis had focused on Uniface because it was a proprietary system, with no full published language specification. This meant that if the methods could be proven on such a language, the the techniques could potentially be extendable to other proprietary 4GLs. The research had also, to a more limited extent explored the potential of using the same methods with the other 4GLs that share Uniface characteristics (such as being XML-like model-driven 4GLs).

This research studied the CBSE methodology which had been used in developing Uniface itself (section 3.3.2)

Therefore the author utilised the CBSE methodology in reverse engineering Uniface ESIS system code syntax (section 3.3). CBSE is an iterative software development life-cycle that allowed this research to reduce the dependency of the parser on the grammar rules (also known as decoupling). This means that changing a grammar rule will only impact its relevant object-oriented method in the generated C# code as shown in appendix C. Using CBSE permitted segmentation of tasks, which in turn allowed the researcher to target Uniface syntax line by line rather than processing lengthy blocks of code at once. The combination of decoupling and parsing individual lines of Uniface code have allowed the grammar rules to get parsed independently of each other.

Uniface was found to export its content into an XML-like file. The file failed to parse either using HTML tools (section 4.4.1) or using common XML parser tools (section 4.4.2). The novel approach adopted was to use an EDOM tool and methods (see section 4.5) to parse and correct the malformed XML syntax. The author investigated the structure of the XML file and restored its schema. The restored schema was visualised in DTD (figure 5.2) and XSD (figure 5.3) formats.

Extracting the schema of the data modelled by Uniface was a crucial step towards implementation, which enabled automation of extracting the ESIS Uniface system developer's code embedded within the XML.

DTD and XSD models were each explored as possible approaches to extract the schema of extracted Uniface XML file. Both succeeded but the latter was chosen as it provided a richer set of data about the schema (as discussed in section 5.2).

The schema validated the structure of the XML file and enabled precise cuts for the 'OCC' nodes within which the Uniface code had been found to be stored (as described in detail in chapter 6).

Studying the extracted Uniface source code syntax allowed the researcher to manually create grammar rules for Uniface. Such rules were not previously available in the public domain for Uniface. These grammar rules were based purely on the ESIS system and so may not be fully representative of the full implementation of Uniface.

Uniface grammar rules were set out as a formal structure using EBNF rules (as described in section 7.1.9) and thus were able to be parsed and prepared for code generation. This was automated using 'Template Meta-Programming' as discussed in sections 7.1.3, and 7.1.8.

The grammar rules allowed for the parser to subsequently generate its AST and automate parsing Uniface code.

With Uniface grammar rules and the XSD schema, it became possible to automatically parse Uniface syntax using the data structures from the XSD. It also allowed the management of an orderly fragmentation of the XML file (necessary due to the large size of the original XML file).

The methodology used to write to the target language was explained in section S.O.L.I.D. Principles 8.1. The selected target language to translate ESIS Uniface code into was C# (section 8.2).

Parser-parser tools such as Gold Parser and ANTLR were each used in turn to generate the parsing table for the bespoke grammar. Sections 8.3.2, and 8.4.1 discussed both tools in detail.

The author concluded (section 8.5) that ANTLR allowed for semantic instructions when writing the grammar whereas Gold Parser did not. In addition, ANTLR output was more descriptive and enabled ATN networks as shown in figure 8.10. These features meant that ANTLR enabled more appropriate output for debugging and diagnosis, and thus was more appropriate for visual analysis of grammar rules.

The methodology designed in this thesis and explained in detail with respect to the Uniface implementation at the University of Essex, was also applied to demonstrate reverse engineering other 4GLs. These included demonstrations of the method on a small subset of other non XML-like model-driven ones as these were readily available), as described in section 9.1.

The same approach was also tested on a bespoke 4GL that was created by the author to better

understand 4GLs, and then reused to test and demonstrate the author's reverse engineering method. This new language ‘Mini-4GL’ was created to list a number of pharmaceuticals into a table. The compiler parsed and generated a valid HTML styled with a valid CSS syntax, section 9.2.

The details of the processes used for reverse-engineering Mini-4GL were set out in section 9.3. The author concluded that the method was successful. Figure 9.8 illustrated the original syntax for Mini-4GL and the reverse engineered HTML syntax as output.

In addition to providing viable code in the output language (C# generally), the author needed to demonstrate that the output code had the same syntax as that of the original source language. To do this a novel method was designed to show syntactical equivalence between the ASTs (section 10.2) in the form of chunking trees (section 10.3). That section also showed the method in detail and how the trees for the source and target language could provide complete, partial or no translation.

This method was then used, for such grammar rules and syntax as had been implemented in this research, to show the comparison runs between Uniface input and C# output for Uniface (section 10.4.1).

The results both for the Informix (section 10.4.2) and Apex (section 10.4.3) and that for the Mini-4GL (section 10.4.4) were provided. Although on a very small scale, in evaluating the results (section 10.5) the author believes that the methods used in this research offers promising avenues to extend the methods demonstrated on the ESIS Uniface system, to other proprietary model-driven 4GLs.

11.2 Answering the Research Questions

11.2.1 Main Research Question

The main research question was:

What methods can be used to automate the process of translating the code written with a 4GL (such as Uniface) as an exemplar) into a different language syntax such as third-generation language syntax, (C# for instance), in the absence of documentation and syntax definition resources?

This research showed that it may not be possible to directly recover the design of a system written in Uniface such as the University of Essex's ESIS system, using the Uniface meta-data database.

However, the methodologies used in this research (CBSE in section 3.3.2) as well as the use of EDOM (section 4.5) were successful in enabling the extraction of the ESIS system's Uniface source code. It also enabled validating and cleaning the XML-like code in a novel approach (section 4.5). The schema was then visualised in XML schema tools such as DTD and XSD. This

method enabled the parser to work on the Uniface novel grammar rules (as created by the author) and translate the syntax of the Uniface code into C#.

These methods were also adapted to work with other 4GLs, as demonstrated using a limited set of grammar rules.

For both Uniface and the other 4GLs, a novel method (chunking trees 10.3) was used to compare the syntax of input with the output to demonstrate complete translation of the syntax.

Some of the methods and the processes of translation as used in this research were able to be automated as set out in 11.1).

11.2.2 Sub-Research Question 1

What development methodologies, that support modular programming and employ reverse engineering techniques, can be applied to 4GL code, and in particular proprietary 4GL code such as Uniface?

This research discussed CBSE in detail in section 3.3 (Using Component-Based Methodologies Towards Reverse Engineering Uniface). This methodology supported incremental development modularity for building and testing Uniface grammar rules. It also provided the core translation modules were used to provide translation based on the Visitor design pattern. In turn this enabled extending from Uniface on to other 4GLs, the classes and functions shown, for example, in appendix H.

CBSE then allowed grammar rules to be implemented incrementally which enabled reversing Uniface grammar and forward engineering the parser at the same time. In addition, the methodology increased the cohesion between classes and decreased coupling, both of which are always desirable in software engineering. It limited the changes required when changes to a grammar rule have necessitated adaptations to the code in a relevant class.

The above shows that the CBSE methodology was found to be a suitable one for the research involving reverse engineering of Uniface and the other 4GLs studied.

11.2.3 Sub-Research Question 2

What parsing techniques may be used to generate a parse table for the syntax of Uniface in particular and other proprietary 4GLs in general, that is described and represented by a set of grammar rules?

This research applied LALR, and LL parsing techniques to generate the parsers and implemented the proof of concept using Gold Parser and ANTLR tools as shown in sections 8.3, and 8.4.1. Gold Parser implementation was found to be more optimised in terms of performance. ANTLR offered a more complex analytical for the structure of the syntax, as illustrated using Augmented Transition Network (ATN) in figure 8.10.

The above shows that suitable parsing techniques have been developed to parse Uniface in the absence of any pre-defined settings being provided by the two tools explored in depth (or any others investigated).

11.3 Summary of Contributions to Knowledge

This section brings together in one place the novel concepts and techniques discussed throughout this thesis.

11.3.1 Applying CBSE Methodology to a Reverse Engineering Model-Driven Proprietary 4GL

This research applied the CBSE methodology (see section 3.3.2) to a XML-like, model-driven proprietary language (Uniface).

This was novel because up to this point, reverse engineering of 4GLs was approached in isolation of the methodology used to reverse engineer those systems. This research utilised CBSE and found it to be advantageous to incrementally build Uniface grammar rules and extend the translator functions.

This was fully described in section 3.3 (Using Component-Based Methodologies Towards Reverse Engineering Uniface).

11.3.2 Utilising EDOM for Processing XML-like Code into a List of Valid XML Files

This research utilised EDOM for segmenting, fragmenting, and splitting the sourced XML-like code into a list of valid XML files.

This was novel because up to this point Uniface XML was not investigated or corrected for processing. XML is commonly corrected before its schema either DTD or XSD can be generated. This researched overcome the inability to parse the original XML, by parsing streams of XML as chains of characters including the obscure ones (named golden characters by Uniface).

This was fully described in section 4.5 (Encapsulated Document Object Model (EDOM)).

11.3.3 Novel Grammar Rules for Uniface

This research created grammar rules for Uniface, as described in chapter 7 (Implementation IV - Grammar Formalism). The set of created grammar rules was necessarily incomplete due to the absence of language specification, and other factors as set out in section 10.4.

The novelty arose from the creation of such grammar rules as there was no published full language specifications for Uniface, either in the academic portals or commercial platforms prior to at the point which this research was published. The grammar rules resulted in a successful translation of a set of Uniface syntax to the selected target language (in the case of this research C#).

This was fully described in chapter 7 (Implementation IV - Grammar Formalism).

11.3.4 Use of Parser-Parser and Parser Generators to Translate a XML-like Proprietary 4GL into a 3GL

The approach, of using parser-parser and parser-generators with LL, and LALR, to formalise the approaches with XML-like, model-driven proprietary languages is a novel approach.

This was novel because up to this point such techniques had only been applied to upgrade the language version or to attempt to translate it into a similar 4GL with a focus on semantics.

This study attempted to use parser generators to provide a mechanism to transform the syntax of proprietary languages into languages, such as C#, which:

- are more familiar to today's generation of software engineers,
- follow modern language development standards,
- have documentation that is publicly available for the language's syntax grammar rules and also semantics,
- offer free licensing arrangements,
- provide a pool of trained experts.

This was fully described in chapter 8 (Using Parser Generators to Implement Uniface Grammar).

11.3.5 Using Chunking Trees to Syntactically Compare Programming Languages

The area of using AST to syntactically compare the results of translating 4GLs into 3GLs is underpopulated. The use of chunking trees (CTs) to compare results was novel. CTs reduced the number of nodes in both source and destination trees without loosing the nodes in subject and the common root that joins them.

This was described in chapter 10 (Results) and specifically in section 10.3 (Using Chunking Trees to Compare ASTs for Source Language and Target Language Syntax).

11.4 How the System Could be Improved/Completed

To bring the proof of concept system provided here up to a usable standard, the following underlying aspects would need to be addressed:-

- A more complete set of grammar rules (section 11.4.1),
- Automating the parts that could be readily automated (section 11.4.2).

11.4.1 More Complete Set of Grammar Rules for Uniface

This proof of concept system has only implemented some of the Uniface grammar rules to demonstrate the viability of the author's method.

More of the grammar rules could be implemented and tested to confirm or otherwise whether the author's method still provides a complete syntactical translation.

This is likely to involve utilising further exemplar case studies, as there is no guarantee that any one system will exercise all of Uniface's grammar.

Since Uniface was a proprietary language, with no published syntax or full specification, it would be virtually impossible to determine at what stage 100% coverage of it's grammar and syntax would have been achieved.

11.4.2 Automation of the System

Figure 11.1 below shows the system's components together with an indication of which were already fully automated, which were non-automated and which have not been automated in this proof of concept system but could potentially be automated, with current technologies.

Fully Automated
This applies to parts of the system which could be run via computer code. There might well be further scope for making that code more efficient, maintainable or less resource intensive.

Can be Automated
This refers to parts of the system which have been manually performed in this thesis's proof of concept delivery, but which could readily be automated using available current technology.

Non-Automatable
This refers to parts of the system which are manually performed in this thesis's proof of concept delivery, but which could not readily be automated using current available technology. It might be possible that Artificial Intelligence (AI) techniques could be successfully applied but this would require significant effort and might not be a viable process.

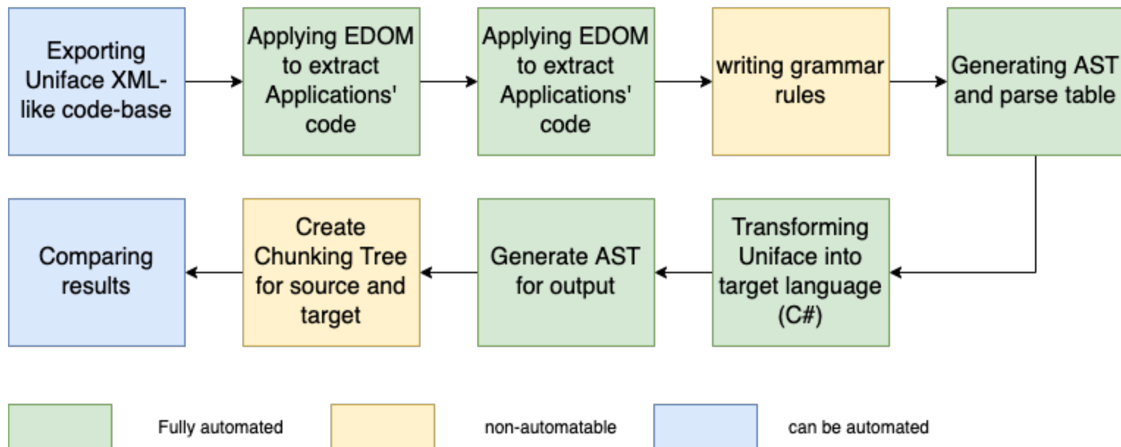

Figure 11.1: Processes Within System of Transforming Uniface by Level of Automation

The levels of automation provided by this proof of concept system were those which could be readily achieved by a single researcher's resources in the time available.

11.5 Lessons Learned

The author of this research would recommend investing in code analysis and estimation tools as was presented in section 2.4.3. This step might be costly and could consume the limited time and resources that were available to conduct the research but it would be valuable to address the proof of concept as an engineering problem with estimations of language scope covered, based on the analysis.

Moreover, the author of this research would have secured more resources to investigate the subject of heterogeneous applications (as discussed in section 2.4.7). This would be possible if it were known at the early stages of this research that a Uniface extract might have multiple languages embodied within.

In addition, the author would have applied 'software slicing' techniques which segments the source code into decoupled segments. This technique is commonly used with tools that study the semantic aspect of languages but could be very helpful even when dealing with syntactical aspects. This study segmented the code using EDOM (section 4.5) and verified it manually but a hybrid approach could enhance the segmentation logic.

11.6 Future Work

This research has introduced a novel method as well as a methodology to target 4GLs that were not yet been targeted for reverse engineering.

In addition to How the System Could be Improved/Completed (section 11.4) above, there are a number of topics that future researchers could usefully explore, as set out below.

- **Reverse engineering of Semantics:** A study to reverse engineer the semantic/behavioural aspects of 4GLs as studied here, would add significant value on top of the syntactical one to enhance the translation and allow for fully functional modules to be translated at once.

  Techniques that might be usefully explored could be:

  - **Software slicing:** This technique could be used to segment the code into functional segment that could run in isolation of each other,
  - **Code analysis and estimation:** Provision of code analysis and estimation could determine the remaining time for the selected modules for translation.

- **Using Natural Language Processing to Reverse Engineer 4GLs:** Natural language processing researchers have used advanced text manipulation techniques to extract knowledge from spoken languages. These techniques might be useful to summarise code by purpose or to assist in reverse engineering the code and enhance existing code translators for 4GLs.

- **Performance analysis and optimisation:** This thesis approached the selected 4GLs for translation not concerned about potential performance degrading.

  Larger scale investigations would be needed to determine whether the method described could be used at scale and what would be needed to make it viable at scale.

11.7 Concluding Remarks

According to Weinberg as cited by Chemuturi [343]:

> "If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilisation."

These words of wisdom apply to computer science in general but particularly to the field of reverse engineering and within that to reverse engineering of proprietary 4GL systems.

This thesis is intended to be a small but significant step towards making such a saying less applicable for this branch of software engineering.

Appendix A

Output Code in C# from Input Code in Uniface

This is the translated code in C# which represents the original Uniface input as provided in appendix B (Input - Uniface Code from ESIS System (Extract)).

```
1  public void LRETRIEVE()
2  {
3  clear(e, "POSTGRADUATE_Obfuscated");
4  //PG_NO.POSTGRADUATE_Obfuscated = "0989898&uALL;" ; T E S T ! ! !
5  POSTGRADUATE_ObfuscatedEntity.YEAR_PROPOSED_ENTR = DUMMY.YEAR;
6  POSTGRADUATE_ObfuscatedEntity.DEPT_IDENT = DUMMY.DEPT;
7  POSTGRADUATE_ObfuscatedEntity.COURSE_IDENT = "{SCHEME.DUMMY}&uALL;";
8  POSTGRADUATE_ObfuscatedEntity.LEVEL_OF_STUDY = DUMMY[1:1].CLASS;
9  POSTGRADUATE_ObfuscatedEntity.STUDY_AIM_TYPE = DUMMY[2:1].CLASS;
10 POSTGRADUATE_ObfuscatedEntity.REG_CATEGORY = DUMMY[3:2].CLASS;
11 POSTGRADUATE_ObfuscatedEntity.MN_SUPP_BODY = DUMMY.SUPP_BODY;
12 if (DUMMY.REPLY_A == "T")
13 {
14 if (DUMMY.REPLY_D == "T")
15 {
16 POSTGRADUATE_Obfuscated.REPLY = "A&uOR;D";
17 }
18 else
19 {
20 POSTGRADUATE_Obfuscated.REPLY = "A";
21 }
22 }
23 else
24 {
```

```
25 if (DUMMY.REPLY_D == "T")
26 {
27 POSTGRADUATE_Obfuscated.REPLY = "D";
28 }
29 }
30
31 //------------
32 if (REPLY_BLANK == "T")
33 {
34 if (REPLY_A = "T" && REPLY_D = "T")
35 if (REPLY_D == "T")
36 {
37 POSTGRADUATE_Obfuscated.REPLY = "A&uOR;D&uOR;&uEQ;";
38 else if (REPLY_A = "T" && REPLY_D != "T")){
39 POSTGRADUATE_Obfuscated.REPLY = "A&uOR;&uEQ;";
40 else if (REPLY_A != "T" && REPLY_D = "T")){
41 POSTGRADUATE_Obfuscated.REPLY = "D&uOR;&uEQ;";
42 }
43 else{
44 POSTGRADUATE_Obfuscated.REPLY = "&uEQ;";
45 }
46 }
47 //-----------------------
48 //PG_NO.POSTGRADUATE_Obfuscated = "0989898";TEST!!!!!!!!!
49 retrieve(e, "POSTGRADUATE_Obfuscated");
50 sort(e, "POSTGRADUATE_Obfuscated", DUMMY.ORDER);
51 //askmess/error "date expected %%DATE_EXPECTED.POSTGRADUATE_Obfuscated%%%  pg-no
   ↪  ;%%PG_NO.POSTGRADUATE_Obfuscated%%%"
52 if (hits(POSTGRADUATE_Obfuscated) != 0){
53 setocc(tocc"POSTGRADUATE_Obfuscated", 1);
54 do
55 {
56 d2 = "";
57 message(Pg_No);
58 //CHECK FEES
59 if (d2 == ""){
60 if (DUMMY.FEE_1 == "T" || DUMMY.FEE_2 == "T" || DUMMY.FEE_3 == "T" || DUMMY.FEE_4 == "T")
61 {
62 if (POSTGRADUATE_APPLICANT.FEE_STATUS == "" )
63 {
64 if (POSTGRADUATE_APPLICANT.FEE_STATUS == "4"){
65 d2 = "d";
66 // Record discarded so no need to increment occurences;
67 }
68 }
69 }
70
71 //CHECK COUNTRIES
72 if (d2 == ""){
```

```
73
74 {
75
76 {
77 d2 = "d";
78 // Record discarded so no need to increment occurences;
79 }
80 }
81 //CHECK ADVERT TYPE &amp; REF NO
82 //   if (d2 = "" &amp; (TYPE_OF_ADVERT.DUMMY != "" | ADVERT_REF_NO.DUMMY != ""))
83 //       clear/e "POSTGRADUATE_ENQUIRY"
84 //       PG_NO.POSTGRADUATE_ENQUIRY = PG_NO.POSTGRADUATE_Obfuscated
85 //       COURSE_IDENT.POSTGRADUATE_ENQUIRY = COURSE_IDENT.POSTGRADUATE_Obfuscated
86 //       YEAR_PROPOSED_ENTRY.POSTGRADUATE_ENQUIRY = YEAR_PROPOSED_ENTR.POSTGRADUATE_Obfuscated
87 //       retrieve/e "POSTGRADUATE_ENQUIRY"
88 //       if ((TYPE_OF_ADVERT.DUMMY != "" &amp; TYPE_OF_ADVERT.DUMMY !=
   ↪  TYPE_OF_ADVERT.PG_ADVERT_DET) | (ADVERT_REF_NO.DUMMY != "" &amp; ADVERT_REF_NO.DUMMY !=
   ↪  ADVERT_REF_NO.PG_ADVERT_DET))
89 //         d2 = "d"  ;Record discarded so no need to increment occurences
90 //      endif
91 //   endif
92 //CHECK ENGLISH LANG QUALS REQUIRED
93
94
95 {
96 if (DUMMY.EL_QUAL_REQD != EL_QUAL_REQ1.PG_EL_REQD_DET && EL_QUAL_REQD.DUMMY) {
97
98 {
99 d2 = "d";
100 }
101 }
102 //CHECK PRE_SESS_COURSE TIME
103
104
105 {
106 if (DUMMY.PRE_SESS != PRE_SESS_COURSE.PG_EL_REQD_DET)
107 {
108 d2 = "d";
109 }
110 }
111 //CHECK SUPERVISOR
112
113
114 {
115 if (DUMMY.SUPERVISOR != SUPERVISOR_KEY1.POSTGRADUATE_Obfuscated &&
116 SUPERVISOR_KEY2.POSTGRADUATE_Obfuscated ){
117
118 {
119 d2 = "d";
```

```
120 }
121 }
122 // CHECK NATURE OF STUDY (taught / research)
123 clear(e, "STUDY_PROGRAMME");
124 STUDY_PROGRAMME.COURSE_IDENT = POSTGRADUATE_Obfuscated[1:10].COURSE_IDENT;
125 retrieve(e, "STUDY_PROGRAMME");
126 if (d2 == "" &&
127 NATURE_T.DUMMY || DUMMY.NATURE_R == "T")
128 {
129 if (DUMMY.NATURE_T == "T" | NATURE_R.DUMMY){
130 if (STUDY_PROGRAMME.NATURE_OF_STUDY == "2")
131 {
132 d2 = "d";
133 }
134
135 if (STUDY_PROGRAMME.NATURE_OF_STUDY == "1")
136 {
137 d2 = "d";
138 }
139 }
140 //CHECK CAMPUS (B,N,G,F,J,L,N,P,R in STUDY_PROGRAMME.   There is also a lookup in CAMPUS)
141 if (d2 = "" && TCAMPUS != "")
142
143 {
144 if (STUDY_PROGRAMME.CAMPUS_ID != TCAMPUS)
145 {
146 d2 = "d";
147 }
148 }
149
150 //CHECK TRANSACTION TYPE &amp; DECISION
151 //POSTGRADUATE_OFFER IS ORDERED BY OFFER_NO DESC IN READ TRIGGER
152 if (d2 == "" && TYPE_LA.DUMMY || DUMMY.TYPE_LD == "T" || DUMMY.TYPE_RD == "T")
153 {
154 if (DUMMY.TYPE_LA == "T" || TYPE_LD.DUMMY){
155 if (POSTGRADUATE_OFFER.TRANS_TYPE == "")
156 {
157 if (POSTGRADUATE_OFFER.TRANS_TYPE == TRANS_TYPE.POSTGRADUATE_OFFER )
158 {
159 d2 = "d";
160 }
161 }
162
163 //----------------
164 if (DUMMY.DECIS_U == "T" || DUMMY.DECIS_C == "T" || DUMMY.DECIS_R == "T" || DUMMY.DECIS_F ==
    ↪ "T")
165 {
166 //askmess/error " statrt d10 is %%d10%%%"
167 if (POSTGRADUATE_OFFER.DECISION == "" || POSTGRADUATE_OFFER.DECISION == "WD")
```

```
168 {
169 if (POSTGRADUATE_OFFER.DECISION == DECISION.POSTGRADUATE_OFFER)
170 {
171 d2 = "d";
172 d10 = d10;
173 }
174 else{
175 d10 = "keep";
176 }
177 compare(t, PG_NO,from);
178 //askmess/error " $result %%$result%%%"
179 if (result < 0 && d10 == "keep")
180 if (d10 == "keep"){
181 d2 = "";
182 d10 = "";
183 }
184 //askmess/error " here 1  choice no %%CHOICE_NO.POSTGRADUATE_Obfuscated%%% at end d2 is
    ↪  %%d2%%% d10 is %%d10%%%"
185 //-------------------------------------------------
186 //---do not include if offer is excluded from batch; - added 30/05/06
187 if (POSTGRADUATE_OFFER.OFFER_LETTER_STATUS == "X") ;
188 excluded{
189 d2 = "d";
190 message( %%SURNAME %%);
191 }
192 //---
193 //---do not include if offer is not approved; - added 329/04/08
194 message( "%%PG_NO.POSTGRADUATE_OFFER %%");
195 if (POSTGRADUATE_OFFER.DECISION == "U" || POSTGRADUATE_OFFER.DECISION == "C")
196 {
197 if (POSTGRADUATE_OFFER.OFFER_LETTER_STATUS != "C") ;
198 // not approved, so discard...{
199 d2 = "d";
200 message(discarded);
201 }
202 }
203 //---
204 }
205 //CHECK START DATE AND END_DATE
206 //askmess/error " testing %%PG_NO.POSTGRADUATE_Obfuscated%%% DATE EXPECTED
    ↪  %%DATE_EXPECTED.POSTGRADUATE_Obfuscated%%%"
207
208 if (POSTGRADUATE_Obfuscated && START_AFTER.DATE_EXPECTED ==
    ↪  DATE_EXPECTED.POSTGRADUATE_Obfuscated && START_AFTER.DUMMY && POSTGRADUATE_Obfuscated &&
    ↪  START_BEFORE.DATE_EXPECTED = "");
209 d2 = "d";
210 }
211 //askmess/error "d2 after date check.......... %%d2%%%"
212 //EARLIEST &amp; LATEST DATE OF OFFER
```

```
213  //askmess/error " testing %%PG_NO.POSTGRADUATE_Obfuscated%%% DATE_OF_OFFER
     ↪  %%DATE_OF_OFFER.POSTGRADUATE_OFFER%%%"
214
215  if (POSTGRADUATE_OFFER &&  TEARLIEST_OFFER_DATE.DATE_OF_OFFER ==
     ↪  DATE_OF_OFFER.POSTGRADUATE_OFFER && TEARLIEST_OFFER_DATE.DUMMY && POSTGRADUATE_OFFER &&
     ↪  TLATEST_OFFER_DATE.DATE_OF_OFFER = "");
216  d2 = "d";
217  }
218  //askmess/error "d2 after date check.......... %%d2%%%"
219  //if Deferred offer (DD) then do not select anyway.
220  if (POSTGRADUATE_Obfuscated.REPLY == "DD")
221  {
222  message(deferred);
223  d2 = "d";
224  }
225
226  //CHECK DISABILITY
227  if (TDISABILITY != "")
228  {
229  if (d2 == ""){
230  if (POSTGRADUATE_APPLICANT.DISABLITY_IND == "0")
231  {
232  d2 = "d";
233  d2 = "d";
234  Record discarded so no need to increment occurences;
235  }
236  }
237  }
238
239  //CHECK EARLIEST CREATE DATE
240  if (TEARLIEST_CR_DATE != "")
241  {
242  if (d2 == ""){
243  d2 = "d";
244  }
245  }
246  }
247
248  //*******************************************************
249  //* For the two repeat loops below if any one pass     *
250  //*proves correct then we don't want to discard hence  *
251  //* the setting of d2 to "" and break .. DP 22/4/03     *
252  //*******************************************************
253  //CHECK ENTRY QUALS INSTITUTION
254  //askmess/error " here 1)"
255
256  {
257
258  {
```

```
259 //askmess/error "here 2)"
260 setocc(setocc, "POSTGRADUATE_QUALS");
261 do
262 {
263 if (DUMMY.PG_INSTIT_CODE != "")
264 {
265 if (DUMMY.PG_INSTIT_CODE != PG_INSTIT_CODE.POSTGRADUATE_QUALS)
266 {
267 d2 = "d";
268 }
269 else
270 {
271 d2 = "";
272 }
273 }
274
275 //askmess/error " d2 is now %%d2%%%"
276 if (DUMMY.QUAL_COUNTRY != "" && PG_INSTIT_CODE.DUMMY) {
277 if (DUMMY.PG_INSTIT_CODE == "") ;
278 ;
279 ;
280 added last bi){
281 if (DUMMY.QUAL_COUNTRY != COUNTRY_ID.POSTGRADUATE_QUALS)
282 {
283 d2 = "d";
284 }
285 else
286 {
287 d2 = "";
288 }
289 }
290 if (d2 == ""){
291 break;
292 }
293 setocc(setocc"POSTGRADUATE_QUALS",
294 curoccPOSTGRADUATE_QUALS)1);
295 until(status >= 0){
296 }
297 //askmess/error " d2 is now(2) %%d2%%%"
298 //CHECK ENGLISH LANG QUALS HELD
299
300
301 {
302 setocc(setocc, "POSTGRADUATE_ENG_LANGQUALS");
303 do
304 {
305 if (DUMMY.EL_QUAL_HELD != EL_QUAL.POSTGRADUATE_ENG_LANG_QUALS)
306 {
307 d2 = "d";
```

```
308 }
309 else
310 {
311 d2 = "";
312 break;
313 }
314
315 setocc(setocc"POSTGRADUATE_ENG_LANGQUALS",
316 curoccPOSTGRADUATE_ENG_LANGQUALS)1);
317 until(status >= 0){
318 }
319 //***Remove duplicates
320 compare(t, PG_NO);
321 d2 = "d";
322 }
323
324 if (d2 == ""){
325 setocc(setocc"POSTGRADUATE_Obfuscated",
326 curoccPOSTGRADUATE_Obfuscated)1);
327 }
328 else{
329 }
330 until(status >= 0){
331 setocc(tocc"POSTGRADUATE_Obfuscated", 1);
332 setocc(tocc"POSTGRADUATE_Obfuscated", 1added);
333 }
334 DUMMY.APPLICATIONS = hits(POSTGRADUATE_Obfuscated);
335 setocc(c"POSTGRADUATE_Obfuscated", 1);
336 //XXXXXXXXxxxxxxxxxxxxxxxxxxxXXX
337 //create the list in dummy4
338 setocc(c"POSTGRADUATE_Obfuscated", 1);
339 setocc(c"POSTGRADUATE_Obfuscated", 1);
340 do
341 {
342 "DUMMY4"
343 DUMMY4.PG_NO2 = POSTGRADUATE_Obfuscated.PG_NO;
344 clear(e, "POSTGRADUATE_ADDRESS");
345 POSTGRADUATE_ADDRESS.PG_NO =
346 POSTGRADUATE_Obfuscated.PG_NO;
347 retrieve(e, "POSTGRADUATE_ADDRESS");
348 if (POSTGRADUATE_ADDRESS.EMAIL_OK == "N")
349 {
350 TEMAIL2 = "";
351 field_video TEMAIL2, "col=52";
352 }
353
354 setocc(occ,"POSTGRADUATE_Obfuscated",
355 curoccPOSTGRADUATE_Obfuscated,1);
356 until(status >= 0){
```

```
357 //
358 if (PG_NO2 == "")
359 {
360 askmess("No);
361 }
362 }
363
364 //
365 //------------------------------------------------------------
366 public void LOLDEMAIL()
367 {
368 string FROMADDRESS,
369 TOADDRESS,
370 CMDLINE,
371 RES,
372 ADDR;
373 }
374
375 askmess("You);
376 if (status != 1){
377 return;
378 }
379 d1 = 0;
380 CT = 0;
381 d99 = 0;
382 //FROMADDRESS = "hagrid@hogwarts.ac.uk"; must be a real address
383 FROMADDRESS = TFROM_TEXT;
384 //askmess/error " $hits %%$hits(POSTGRADUATE_Obfuscated)%%%"
385 setocc(c"POSTGRADUATE_Obfuscated", 1);
386 setocc(c"POSTGRADUATE_Obfuscated", 1);
387 setocc(c"DUMMY4", 1);
388 setocc(c"DUMMY4", 1);
389 //;;;setocc "POSTGRADUATE_Obfuscated",1
390 message( "%%PG_NO %%");
391 while (status){
392 if (TEMAIL2 != "")
393 {
394 TOADDRESS = "{TEMAIL2}%";
395 d1 = d1 + 1;
396 message(Record);
397 //askmess/error "CMDLINE %%CMDLINE%%%"
398 activate();
399 if (RES != "")
400 {
401 askmess("Possible);
402 if (status == 1){
403 activate();
404 if (RES == "")
405 {
```

```
406 message(mail);
407 }
408 }
409 }
410 }
411 
412 setocc(occ,"DUMMY4", curoccDUMMY4),1);
413 }
414 }
415 
416 //-------------------------------------------------------
417 public void LGET_FROM_ADDRESS()
418 {
419 //** needs to create a from address based on the users logon id.
420 if (USERNAME == ""){
421 USERNAME = "hermione";
422 }
423 }
```

Appendix B

Input - Uniface Code from ESIS System (Extract)

This piece of code was an extract of Uniface taken from a sample 'Table' element discussed in section 6.1.3.2.

Some of the data has been omitted or obfuscated to avoid privacy issues arising.

```
entry LRETRIEVE
  clear/e "POSTGRADUATE_Obfuscated"
  ;PG_NO.POSTGRADUATE_Obfuscated = "0989898&uALL;" ; T E S T ! ! !
  YEAR_PROPOSED_ENTR.POSTGRADUATE_Obfuscated = YEAR.DUMMY
  DEPT_IDENT.POSTGRADUATE_Obfuscated = DEPT.DUMMY
  COURSE_IDENT.POSTGRADUATE_Obfuscated = "%%SCHEME.DUMMY%%&uALL;"
  LEVEL_OF_STUDY.POSTGRADUATE_Obfuscated = CLASS.DUMMY[1:1]
  STUDY_AIM_TYPE.POSTGRADUATE_Obfuscated = CLASS.DUMMY[2:1]
  REG_CATEGORY.POSTGRADUATE_Obfuscated = CLASS.DUMMY[3:2]
  MN_SUPP_BODY.POSTGRADUATE_Obfuscated = SUPP_BODY.DUMMY
  if (REPLY_A.DUMMY = "T")
     if (REPLY_D.DUMMY = "T")
        REPLY.POSTGRADUATE_Obfuscated = "A&uOR;D"
     else
        REPLY.POSTGRADUATE_Obfuscated = "A"
     endif
  else
     if (REPLY_D.DUMMY = "T")
        REPLY.POSTGRADUATE_Obfuscated = "D"
```

```
20        endif
21     endif
22     ;------------
23     if (REPLY_BLANK ="T")
24       if (REPLY_A = "T" &amp; REPLY_D = "T")
25          REPLY.POSTGRADUATE_Obfuscated = "A&uOR;D&uOR;&uEQ;"
26       elseif (REPLY_A = "T" &amp; REPLY_D != "T")
27          REPLY.POSTGRADUATE_Obfuscated = "A&uOR;&uEQ;"
28       elseif (REPLY_A != "T" &amp; REPLY_D = "T")
29          REPLY.POSTGRADUATE_Obfuscated = "D&uOR;&uEQ;"
30       else
31          REPLY.POSTGRADUATE_Obfuscated = "&uEQ;"
32       endif
33     endif
34     ;-----------------------
35
36
37
38     ;PG_NO.POSTGRADUATE_Obfuscated = "0989898";TEST!!!!!!!!!
39     retrieve/e "POSTGRADUATE_Obfuscated"
40     sort/e "POSTGRADUATE_Obfuscated", ORDER.DUMMY
41
42     ;askmess/error "date expected %%DATE_EXPECTED.POSTGRADUATE_Obfuscated%%%
       ↪  pg-no ;%%PG_NO.POSTGRADUATE_Obfuscated%%%"
43
44     if ($hits(POSTGRADUATE_Obfuscated) != 0)
45        setocc "POSTGRADUATE_Obfuscated", 1
46        repeat
47           $2 = ""
48           message "Finding Pg_No %%PG_NO.POSTGRADUATE_Obfuscated%%%....."
49           ;CHECK FEES
50
51           if ($2 = "")
52              if (FEE_1.DUMMY = "T" | FEE_2.DUMMY = "T" | FEE_3.DUMMY = "T" |
                ↪  FEE_4.DUMMY = "T")
```

```
53              if ((FEE_STATUS.POSTGRADUATE_APPLICANT = "") |(FEE_1.DUMMY !=
                ↪  "T" &amp; FEE_STATUS.POSTGRADUATE_APPLICANT =
                ↪  "1")|(FEE_2.DUMMY != "T" &amp;
                ↪  FEE_STATUS.POSTGRADUATE_APPLICANT = "2")|(FEE_3.DUMMY !=
                ↪  "T" &amp; FEE_STATUS.POSTGRADUATE_APPLICANT =
                ↪  "3")|(FEE_4.DUMMY != "T" &amp;
                ↪  FEE_STATUS.POSTGRADUATE_APPLICANT = "4"))
54                 $2 = "d"  ;Record discarded so no need to increment
                   ↪  occurences
55              endif
56           endif
57        endif
58
59        ;CHECK COUNTRIES
60
61        if ($2 = "")
62           if ((C_RESIDENCE.DUMMY != "" &amp; C_RESIDENCE.DUMMY !=
             ↪  C_RESIDENCE.POSTGRADUATE_APPLICANT)|(C_BIRTH.DUMMY != ""
             ↪  &amp; C_BIRTH.DUMMY !=
             ↪  C_BIRTH.POSTGRADUATE_APPLICANT)|(C_NATIONALITY.DUMMY != ""
             ↪  &amp; C_NATIONALITY.DUMMY !=
             ↪  C_NATIONALITY.POSTGRADUATE_APPLICANT))
63              $2 = "d"  ;Record discarded so no need to increment
                ↪  occurences
64           endif
65        endif
66
67        ;CHECK ADVERT TYPE &amp; REF NO
68
69     ;   if ($2 = "" &amp; (TYPE_OF_ADVERT.DUMMY != "" |
       ↪  ADVERT_REF_NO.DUMMY != ""))
70     ;      clear/e "POSTGRADUATE_ENQUIRY"
71     ;      PG_NO.POSTGRADUATE_ENQUIRY = PG_NO.POSTGRADUATE_Obfuscated
72     ;      COURSE_IDENT.POSTGRADUATE_ENQUIRY =
       ↪  COURSE_IDENT.POSTGRADUATE_Obfuscated
73     ;      YEAR_PROPOSED_ENTRY.POSTGRADUATE_ENQUIRY =
       ↪  YEAR_PROPOSED_ENTR.POSTGRADUATE_Obfuscated
74     ;      retrieve/e "POSTGRADUATE_ENQUIRY"
```

```
75      ;       if ((TYPE_OF_ADVERT.DUMMY != "" &amp; TYPE_OF_ADVERT.DUMMY !=
        ↪  TYPE_OF_ADVERT.PG_ADVERT_DET) | (ADVERT_REF_NO.DUMMY != "" &amp;
        ↪  ADVERT_REF_NO.DUMMY != ADVERT_REF_NO.PG_ADVERT_DET))
76       ;         $2 = "d"  ;Record discarded so no need to increment
         ↪  occurences
77       ;      endif
78       ;  endif
79
80
81         ;CHECK ENGLISH LANG QUALS REQUIRED
82
83         if ($2 = "" &amp; EL_QUAL_REQD.DUMMY != "")
84            if ((EL_QUAL_REQD.DUMMY != EL_QUAL_REQ1.PG_EL_REQD_DET) &amp;
              ↪  (EL_QUAL_REQD.DUMMY != EL_QUAL_REQ2.PG_EL_REQD_DET) &amp;
              ↪  (EL_QUAL_REQD.DUMMY != EL_QUAL_REQ3.PG_EL_REQD_DET))
85               $2 = "d"
86            endif
87         endif
88
89         ;CHECK PRE_SESS_COURSE TIME
90
91         if ($2 = "" &amp; PRE_SESS.DUMMY != "")
92            if (PRE_SESS.DUMMY != PRE_SESS_COURSE.PG_EL_REQD_DET)
93               $2 = "d"
94            endif
95         endif
96
97
98
99         ;CHECK SUPERVISOR
100
101        if ($2 = "" &amp; SUPERVISOR.DUMMY != "")
102           if ((SUPERVISOR.DUMMY !=
              ↪  SUPERVISOR_KEY1.POSTGRADUATE_Obfuscated) &amp;
              ↪  (SUPERVISOR_KEY2.POSTGRADUATE_Obfuscated !=
              ↪  SUPERVISOR.DUMMY))
103              $2 = "d"
104           endif
105        endif
```

```
106
107
108          ; CHECK NATURE OF STUDY (taught / research)
109         clear/e "STUDY_PROGRAMME"
110         COURSE_IDENT.STUDY_PROGRAMME =
            ↪  COURSE_IDENT.POSTGRADUATE_Obfuscated[1:10]
111         retrieve/e "STUDY_PROGRAMME"
112         if ($2 = "" &amp; (NATURE_T.DUMMY = "T" | NATURE_R.DUMMY = "T"))
113             if (NATURE_T = "T" &amp; NATURE_OF_STUDY.STUDY_PROGRAMME = "2")
114               $2 = "d"
115             endif
116             if (NATURE_R = "T" &amp; NATURE_OF_STUDY.STUDY_PROGRAMME = "1")
117               $2 = "d"
118             endif
119          endif
120
121
122          ;CHECK CAMPUS (B,N,G,F,J,L,N,P,R in STUDY_PROGRAMME.   There is
             ↪  also a lookup in CAMPUS)
123          if ($2 = "" &amp; TCAMPUS != "")
124            if (CAMPUS_ID.STUDY_PROGRAMME != TCAMPUS)
125              $2 = "d"
126            endif
127          endif
128
129
130          ;CHECK TRANSACTION TYPE &amp; DECISION
131          ;POSTGRADUATE_OFFER IS ORDERED BY OFFER_NO DESC IN READ TRIGGER
132
133          if ($2 = "" &amp; (TYPE_LA.DUMMY = "T" | TYPE_LD.DUMMY = "T" |
             ↪  TYPE_RD.DUMMY = "T" ))
134             if ((TRANS_TYPE.POSTGRADUATE_OFFER = "") |
                ↪  (TRANS_TYPE.POSTGRADUATE_OFFER = "LA" &amp; TYPE_LA.DUMMY !=
                ↪  "T") | (TRANS_TYPE.POSTGRADUATE_OFFER = "LD" &amp;
                ↪  TYPE_LD.DUMMY != "T") | (TRANS_TYPE.POSTGRADUATE_OFFER =
                ↪  "RD" &amp; TYPE_RD.DUMMY != "T"))
135               $2 = "d"
136             endif
137          endif
```

```
138
139     ;----------------
140
141         if ((DECIS_U.DUMMY = "T" | DECIS_C.DUMMY = "T" | DECIS_R.DUMMY = "T"
         ↪   | DECIS_F.DUMMY = "T" ))
142             ;askmess/error " statrt $10 is %%$10%%%"
143             if ((DECISION.POSTGRADUATE_OFFER = "") |
             ↪   (DECISION.POSTGRADUATE_OFFER = "U" &amp; DECIS_U.DUMMY !=
             ↪   "T") | (DECISION.POSTGRADUATE_OFFER = "C" &amp; DECIS_C.DUMMY
             ↪   != "T") | (DECISION.POSTGRADUATE_OFFER[1:1] = "R" &amp;
             ↪   DECIS_R.DUMMY != "T") | (DECISION.POSTGRADUATE_OFFER = "F"
             ↪   &amp; DECIS_F.DUMMY != "T") | (DECISION.POSTGRADUATE_OFFER =
             ↪   "WD" ))
144               $2 = "d"
145               $10 = $10
146             else
147               $10 ="keep"
148              endif
149             compare/next (PG_NO)from "POSTGRADUATE_Obfuscated"
150             ;askmess/error " $result %%$result%%%"
151             if ($result &lt;0 &amp; $10 = "keep")
152               $2 = ""
153               $10 = ""
154             endif
155              ;askmess/error " here 1  choice no
              ↪   %%CHOICE_NO.POSTGRADUATE_Obfuscated%%% at end $2 is %%$2%%%
              ↪   $10 is %%$10%%%"
156
157     ;--------------------------------------------------
158
159
160              ;---do not include if offer is excluded from batch; - added
              ↪   30/05/06
161              if (OFFER_LETTER_STATUS.POSTGRADUATE_OFFER = "X") ; excluded
162                $2 = "d"
163                 message "%%PG_NO.POSTGRADUATE_OFFER%%%  %%SURNAME%%% excluded
                  ↪   from batch - discarded"
164              endif
165              ;---
```

```
166            ;---do not include if offer is not approved; - added 329/04/08
167            message " %%PG_NO.POSTGRADUATE_OFFER%%% Letter flag is
             ↪  %%OFFER_LETTER_STATUS.POSTGRADUATE_OFFER%%% decision is
             ↪  %%DECISION.POSTGRADUATE_OFFER%%%"
168            if (DECISION.POSTGRADUATE_OFFER = "U" |
             ↪  DECISION.POSTGRADUATE_OFFER = "C")
169                if (OFFER_LETTER_STATUS.POSTGRADUATE_OFFER != "C"); not
                 ↪  approved, so discard...
170                  $2 = "d"
171                 message"%%PG_NO.POSTGRADUATE_OFFER%%%  %%SURNAME%%%...
                  ↪  discarded:- not approved"
172                endif
173             endif
174             ;---
175          endif
176
177         ;CHECK START DATE AND END_DATE
178         ;askmess/error " testing %%PG_NO.POSTGRADUATE_Obfuscated%%% DATE
          ↪  EXPECTED %%DATE_EXPECTED.POSTGRADUATE_Obfuscated%%%"
179         if ($2 = "" &amp; ((DATE_EXPECTED.POSTGRADUATE_Obfuscated &lt;
          ↪  START_AFTER.DUMMY &amp; START_AFTER.DUMMY != "") %\
180    |(DATE_EXPECTED.POSTGRADUATE_Obfuscated &gt; START_BEFORE.DUMMY &amp;
     ↪  START_BEFORE.DUMMY != ""))
181            $2 = "d"
182         endif
183    ;askmess/error "$2 after date check......... %%$2%%%"
184
185         ;EARLIEST &amp; LATEST DATE OF OFFER
186    ;askmess/error " testing %%PG_NO.POSTGRADUATE_Obfuscated%%% DATE_OF_OFFER
     ↪  %%DATE_OF_OFFER.POSTGRADUATE_OFFER%%%"
187          if ($2 = "" &amp; ((DATE_OF_OFFER.POSTGRADUATE_OFFER &lt;
           ↪  TEARLIEST_OFFER_DATE.DUMMY &amp; TEARLIEST_OFFER_DATE.DUMMY !=
           ↪  "") %\
188    |(DATE_OF_OFFER.POSTGRADUATE_OFFER &gt; TLATEST_OFFER_DATE.DUMMY &amp;
     ↪  TLATEST_OFFER_DATE.DUMMY != ""))
189            $2 = "d"
190         endif
191    ;askmess/error "$2 after date check......... %%$2%%%"
192
```

```
193         ;if Deferred offer (DD) then do not select anyway.
194         if (REPLY.POSTGRADUATE_Obfuscated = "DD")
195            message "%%PG_NO.POSTGRADUATE_Obfuscated%%% deferred. -
               ↪  discarded"
196            $2 = "d"
197         endif
198
199
200        ;CHECK DISABILITY
201        if (TDISABILITY != "")
202           if ($2 = "")
203              if (TDISABILITY = "ALL" &amp;
                 ↪  DISABLITY_IND.POSTGRADUATE_APPLICANT = "O")
204                 $2 = "d"
205              elseif (TDISABILITY != "ALL" &amp; TDISABILITY.DUMMY !=
                 ↪  DISABLITY_IND.POSTGRADUATE_APPLICANT)
206                 $2 = "d"  ;Record discarded so no need to increment
                    ↪  occurences
207              endif
208           endif
209        endif
210
211        ;CHECK EARLIEST CREATE DATE
212        if (TEARLIEST_CR_DATE != "")
213           if ($2 = "")
214             if (CREATE_DATE.POSTGRADUATE_APPLICANT &lt; TEARLIEST_CR_DATE)
215               $2 = "d"
216             endif
217           endif
218        endif
219
220
221
222    ;******************************************************
223    ;* For the two repeat loops below if any one pass     *
224    ;*proves correct then we don't want to discard hence  *
225    ;* the setting of $2 to "" and break .. DP 22/4/03    *
226    ;******************************************************
227          ;CHECK ENTRY QUALS INSTITUTION
```

```
228     ;askmess/error " here 1)"
229           if ($2 = "" &amp; (PG_INSTIT_CODE.DUMMY != "" | QUAL_COUNTRY.DUMMY
          ↪  != ""))
230     ;askmess/error "here 2)"
231              setocc "POSTGRADUATE_QUALS" 1
232              repeat
233                 if (PG_INSTIT_CODE.DUMMY != "")
234                   if (PG_INSTIT_CODE.DUMMY !=
                  ↪  PG_INSTIT_CODE.POSTGRADUATE_QUALS)
235                      $2 = "d"
236                   else
237                      $2 = ""
238                   endif
239                 endif
240     ;askmess/error " $2 is now %%$2%%%"
241                 if (QUAL_COUNTRY.DUMMY != ""  &amp; PG_INSTIT_CODE.DUMMY =
                ↪  "");;;added last bit
242                   if (QUAL_COUNTRY.DUMMY != COUNTRY_ID.POSTGRADUATE_QUALS)
243                     $2 = "d"
244                   else
245                      $2 = ""
246                   endif
247                 endif
248                 if ($2 = "")
249                  break
250                 endif
251                 setocc "POSTGRADUATE_QUALS", $curocc(POSTGRADUATE_QUALS) + 1
252              until ($status &lt;= 0)
253           endif
254     ;askmess/error " $2 is now(2) %%$2%%%"
255
256           ;CHECK ENGLISH LANG QUALS HELD
257
258           if ($2 = "" &amp; EL_QUAL_HELD.DUMMY != "")
259              setocc "POSTGRADUATE_ENG_LANG_QUALS" 1
260              repeat
261                 if (EL_QUAL_HELD.DUMMY !=
                ↪  EL_QUAL.POSTGRADUATE_ENG_LANG_QUALS)
262                    $2 = "d"
```

```
263                 else
264                    $2 = ""
265                    break
266                 endif
267                 setocc "POSTGRADUATE_ENG_LANG_QUALS",
                     ↪  $curocc(POSTGRADUATE_ENG_LANG_QUALS) + 1
268              until ($status &lt;= 0)
269           endif
270
271
272     ;***Remove duplicates
273     compare/next PG_NO from "POSTGRADUATE_Obfuscated"
274     if ($result &gt;0)
275       $2 = "d"
276     endif
277
278
279
280
281           if ($2 = "")
282              setocc "POSTGRADUATE_Obfuscated",
                  ↪  $curocc(POSTGRADUATE_Obfuscated) + 1
283           else
284              discard "POSTGRADUATE_Obfuscated"
285           endif
286        until ($status &lt;= 0)
287        setocc "POSTGRADUATE_Obfuscated",-1
288        setocc "POSTGRADUATE_Obfuscated", 1;added
289     endif
290     APPLICATIONS.DUMMY = $hits(POSTGRADUATE_Obfuscated)
291     setocc "POSTGRADUATE_Obfuscated",-1
292     ;XXXXXXXXxxxxxxxxxxxxxxxxxxXXX
293     ;create the list in dummy4
294     setocc "POSTGRADUATE_Obfuscated",-1
295     setocc"POSTGRADUATE_Obfuscated",1
296     repeat
297       creocc "DUMMY4"
298       PG_NO2.DUMMY4 = PG_NO.POSTGRADUATE_Obfuscated
299       DEPT2 = DEPT_IDENT.POSTGRADUATE_Obfuscated
```

```
300       SURNAME2 = SURNAME.POSTGRADUATE_APPLICANT
301       OTHER_NAMES2 = OTHER_NAMES.POSTGRADUATE_APPLICANT
302       clear/e "POSTGRADUATE_ADDRESS"
303       PG_NO.POSTGRADUATE_ADDRESS = PG_NO.POSTGRADUATE_Obfuscated
304       retrieve/e "POSTGRADUATE_ADDRESS"
305       TEMAIL2 = E_MAIL.POSTGRADUATE_ADDRESS
306       if (EMAIL_OK.POSTGRADUATE_ADDRESS = "N")
307               TEMAIL2= ""
308               field_video TEMAIL2,"col=52"
309             endif
310       setocc "POSTGRADUATE_Obfuscated",$curocc(POSTGRADUATE_Obfuscated) + 1
311     until ($status &lt;=0)
312
313     ;
314
315     if (PG_NO2 = "")
316       askmess/info "No data retrieved for that selection","OK"
317     endif
318
319
320   end; LRETRIEVE
321   ;
322   ;-------------------------------------------------------------
323   Entry LOLDEMAIL ;stand by
324     variables
325       string  FROMADDRESS, TOADDRESS, CMDLINE, RES, ADDR
326       numeric CT
327
328     endvariables
329
330     askmess "You are about to email all displayed applicants.%%^Are you sure
       ↪  you wish to continue?"
331     if ($status != 1)
332       return
333     endif
334     $1 = 0
335     CT = 0
336     $99= 0
337     ;FROMADDRESS = "hagrid@hogwarts.ac.uk"; must be a real address
```

```
338     FROMADDRESS = TFROM_TEXT
339     ;askmess/error " $hits %%$hits(POSTGRADUATE_Obfuscated)%%%"
340     setocc "POSTGRADUATE_Obfuscated", 1
341     setocc "POSTGRADUATE_Obfuscated", -1
342     setocc "DUMMY4", -1
343     setocc "DUMMY4", 1
344     ;;;;setocc "POSTGRADUATE_Obfuscated",1
345     message " %%PG_NO%%% $status %%$status%%%"
346     while ($status &gt;0)
347       if (TEMAIL2!= "")
348         TOADDRESS = "%%TEMAIL2%%%"
349         $1 = $1 + 1
350         message " Record %%$1%%%.    Sending to ... %%TOADDRESS%%% "
351         CMDLINE = "p:\utilities\mailer.exe %%"c:\log\messages.txt%%" -server
          ↪  %%"smtp.office365.com%%" -subject %%"Message from University
          ↪  POSTGRADUATEuate Admissions%%" -f %%"%%FROMADDRESS%%%%%" -to
          ↪  %%"%%TOADDRESS%%%%%" -q"
352      ;askmess/error "CMDLINE %%CMDLINE%%%"
353      activate "OSCMD".SYSTEMI(CMDLINE,RES)
354         if(RES != "")
355           askmess "Possible mail failure %%toaddress%%% Try Again?"
356           if ($status = 1)
357             activate "OSCMD".SYSTEMI(CMDLINE,RES)
358             if(RES = "")
359               message/error "Possible mail failure sorry won't Try Again"
360             endif
361           endif
362         endif
363       endif
364       setocc "DUMMY4", $curocc(DUMMY4) + 1
365     endwhile
366   end ; LOLDEmail
367   ;-------------------------------------------------------
368   entry LGET_FROM_ADDRESS
369     ;** needs to create a from address based on the users logon id.
370     if ($$USERNAME = "")
371       $$USERNAME = "hermione"
372     endif
373     TFROM_TEXT = "%%$$USERNAME%%%@hogwarts.ac.uk"
```

```
374
375 end;LGET_FROM_ADDRESS
```

Appendix C

C# Code to Transpile Uniface into C#

```
1               public override object VisitFunctionRule([NotNull]
                ↪  HelloParser.FunctionRuleContext context)
2               {
3                   return base.VisitFunctionRule(context);
4               }
5
6               public override object VisitIfRule([NotNull]
                ↪  HelloParser.IfRuleContext context)
7               {
8                   sb.Append($"if(");
9                   Visit(context.expr());
10                  sb.Append(") {");
11                  Visit(context.statements());
12                  sb.Append("}");
13                  foreach (var item in context.elseifRule())
14                  {
15                      Visit(item);
16                  }
17                  if(context.elseRule() != null)
18                      Visit(context.elseRule());
19
20                  return null;
21              }
```

```
22
23          public override object VisitElseifRule([NotNull]
             ↪  HelloParser.ElseifRuleContext context)
24          {
25              sb.Append("else if(");
26              Visit(context.expr());
27              sb.Append(") {");
28              Visit(context.statements());
29              sb.Append("}");
30              return null;
31          }
32
33          public override object VisitElseRule([NotNull]
             ↪  HelloParser.ElseRuleContext context)
34          {
35              sb.Append("else {");
36              Visit(context.statements());
37              sb.Append("}");
38              return null;
39          }
40
41          public override object VisitAssignemntStatement([NotNull]
             ↪  HelloParser.AssignemntStatementContext context)
42          {
43              string var_name = "var" +
                 ↪  context.VARIABLE().GetText().Substring(1);
44
45              sb.Append(var_name);
46              sb.Append("=");
47              Visit(context.expr());
48              return null;
49          }
50
51          public override object VisitEqualsExpression([NotNull]
             ↪  HelloParser.EqualsExpressionContext context)
52          {
53              Visit(context.expr()[0]);
54              sb.Append("==");
55              Visit(context.expr()[1]);
```

```
56                return null;
57            }
58
59            public override object VisitNotEqualsExpression([NotNull]
              ↪   HelloParser.NotEqualsExpressionContext context)
60            {
61                Visit(context.expr()[0]);
62                sb.Append("!=");
63                Visit(context.expr()[1]);
64                return null;
65            }
66
67            public override object VisitAddExpression([NotNull]
              ↪   HelloParser.AddExpressionContext context)
68            {
69                Visit(context.expr()[0]);
70                sb.Append('+');
71                Visit(context.expr()[1]);
72                return null;
73            }
74
75            public override object VisitParenthesizedExpression([NotNull]
              ↪   HelloParser.ParenthesizedExpressionContext context)
76            {
77                sb.Append("(");
78                Visit(context.expr());
79                sb.Append(")");
80                return null;
81            }
82
83            public override object VisitLiteral([NotNull]
              ↪   HelloParser.LiteralContext context)
84            {
85                sb.Append(context.GetText());
86                return null;
87            }
```

Appendix D

Generated Parsing Table in Gold Parser

```
============================================================
GOLD Parser Builder
Version 5.2.0.
============================================================


========================================
Grammar
========================================


"Name"     = 'Uniface'
"Author"   = 'Majd Yafi'
"Version" = '1.0'
"About"    = 'Uniface Grammar'

"Case Sensitive" = False
"Start Symbol"    = <Statements>

!clear/e "SPECIAL_FEE_REASON"
!retrieve/e "SPECIAL_FEE_REASON"
!if (TSORT = "CODE")
!     sort/e "SPECIAL_FEE_REASON","REASON_CODE"
```

```
24 !  else
25 !     if (TSORT = "DESC")
26 !       sort/e "SPECIAL_FEE_REASON","REASON_TEXT"
27 !    endif
28 !endif
29
30
31
32
33 {String Ch 1} = {Printable} - ['']
34 {String Ch 2} = {Printable} - ["]
35
36 Id             = {Letter}{AlphaNumeric}*
37
38 ! String allows either single or double quotes
39
40 StringLiteral =  '"'  {String Ch 2}* '"'
41
42
43 NumberLiteral = {Digit}+('.'{Digit}+)?
44
45 <Statements> ::= <Statement> <Statements>
46                | <Statement>
47
48 <Statement>  ::=  <BuiltInStatement> | <If>
49  <BuiltInStatement>   ::= <Clear> | <Retrieve> | <Sort>
50 <Clear> ::= clear <ClearParam> StringLiteral
51 <ClearParam> ::= '/e' | <>
52
53 <Retrieve> ::= retrieve <RetrieveParam>   StringLiteral
54 <RetrieveParam> ::= '/e' | <>
55
56 <Sort> ::= sort <RetrieveParam>   <SortValue>
57 <SortValue>::= StringLiteral | StringLiteral ',' <SortValue>
58
59 <SortParam> ::= '/e' | <>
60
61 <If> ::= if '(' ID <Compare> StringLiteral ')' <Statements>  <Else> endif
62
```

```
63  <Else> ::= else <Statements> | <>
64
65  <Compare>::= '>' | '<'  | '<='  | '>=' | '<>' | '='
66
67  <Expression>   ::= <Expression> '>'  <Add Exp>
68                   | <Expression> '<'  <Add Exp>
69                   | <Expression> '<=' <Add Exp>
70                   | <Expression> '>=' <Add Exp>
71                   | <Expression> '==' <Add Exp>
72                   | <Expression> '<>' <Add Exp>
73                   | <Add Exp>
74
75  <Add Exp>      ::= <Add Exp> '+' <Mult Exp>
76                   | <Add Exp> '-' <Mult Exp>
77                   | <Add Exp> '&' <Mult Exp>
78                   | <Mult Exp>
79
80  <Mult Exp>    ::= <Mult Exp> '*' <Negate Exp>
81                  | <Mult Exp> '/' <Negate Exp>
82                  | <Negate Exp>
83
84  <Negate Exp>  ::= '-' <Value>
85                  | <Value>
86
87  <Value>       ::= ID
88                  | StringLiteral
89                  | NumberLiteral
90                  | '(' <Expression> ')'
91
92
93  ========================================
94  Defined Sets
95  ========================================
96
97  {String Ch 1}
    ↪  {Space}!"#$%&()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
98  {String Ch 2}
    ↪  {Space}!#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
99
```

```
100
101  ========================================
102  Terminals
103  ========================================
104
105   0       (EOF)
106   1       (Error)
107   2       Whitespace
108   3       '-'
109   4       '&'
110   5       '('
111   6       ')'
112   7       '*'
113   8       ','
114   9       '/'
115   10      '/e'
116   11      '+'
117   12      '<'
118   13      '<='
119   14      '<>'
120   15      '='
121   16      '=='
122   17      '>'
123   18      '>='
124   19      clear
125   20      else
126   21      endif
127   22      Id
128   23      if
129   24      NumberLiteral
130   25      retrieve
131   26      sort
132   27      StringLiteral
133
134
135  ========================================
136  Nonterminals
137  ========================================
138
```

```
139 28      <Add Exp>
140 29      <BuiltInStatement>
141 30      <Clear>
142 31      <ClearParam>
143 32      <Compare>
144 33      <Else>
145 34      <Expression>
146 35      <If>
147 36      <Mult Exp>
148 37      <Negate Exp>
149 38      <Retrieve>
150 39      <RetrieveParam>
151 40      <Sort>
152 41      <SortParam>
153 42      <SortValue>
154 43      <Statement>
155 44      <Statements>
156 45      <Value>
157
158
159 ========================================
160 Rules
161 ========================================
162
163 0       <Statements> ::= <Statement> <Statements>
164 1       <Statements> ::= <Statement>
165 2       <Statement> ::= <BuiltInStatement>
166 3       <Statement> ::= <If>
167 4       <BuiltInStatement> ::= <Clear>
168 5       <BuiltInStatement> ::= <Retrieve>
169 6       <BuiltInStatement> ::= <Sort>
170 7       <Clear> ::= clear <ClearParam> StringLiteral
171 8       <ClearParam> ::= '/e'
172 9       <ClearParam> ::=
173 10      <Retrieve> ::= retrieve <RetrieveParam> StringLiteral
174 11      <RetrieveParam> ::= '/e'
175 12      <RetrieveParam> ::=
176 13      <Sort> ::= sort <RetrieveParam> <SortValue>
177 14      <SortValue> ::= StringLiteral
```

```
178 15        <SortValue> ::= StringLiteral ',' <SortValue>
179 16        <SortParam> ::= '/e'
180 17        <SortParam> ::=
181 18        <If> ::= if '(' Id <Compare> StringLiteral ')' <Statements> <Else>
    ↪   endif
182 19        <Else> ::= else <Statements>
183 20        <Else> ::=
184 21        <Compare> ::= '>'
185 22        <Compare> ::= '<'
186 23        <Compare> ::= '<='
187 24        <Compare> ::= '>='
188 25        <Compare> ::= '<>'
189 26        <Compare> ::= '='
190 27        <Expression> ::= <Expression> '>' <Add Exp>
191 28        <Expression> ::= <Expression> '<' <Add Exp>
192 29        <Expression> ::= <Expression> '<=' <Add Exp>
193 30        <Expression> ::= <Expression> '>=' <Add Exp>
194 31        <Expression> ::= <Expression> '==' <Add Exp>
195 32        <Expression> ::= <Expression> '<>' <Add Exp>
196 33        <Expression> ::= <Add Exp>
197 34        <Add Exp> ::= <Add Exp> '+' <Mult Exp>
198 35        <Add Exp> ::= <Add Exp> '-' <Mult Exp>
199 36        <Add Exp> ::= <Add Exp> '&' <Mult Exp>
200 37        <Add Exp> ::= <Mult Exp>
201 38        <Mult Exp> ::= <Mult Exp> '*' <Negate Exp>
202 39        <Mult Exp> ::= <Mult Exp> '/' <Negate Exp>
203 40        <Mult Exp> ::= <Negate Exp>
204 41        <Negate Exp> ::= '-' <Value>
205 42        <Negate Exp> ::= <Value>
206 43        <Value> ::= Id
207 44        <Value> ::= StringLiteral
208 45        <Value> ::= NumberLiteral
209 46        <Value> ::= '(' <Expression> ')'
210
211
212 ========================================
213 DFA States
214 ========================================
215
```

```
216 State 0
217         Goto 1      &09 .. &0D, &20, &85, &A0, &1680, &180E, &2000 ..
    ↪ &200A, &2026, &2028, &2029, &202F, &205F, &3000
218         Goto 2      -
219         Goto 3      &
220         Goto 4      (
221         Goto 5      )
222         Goto 6      *
223         Goto 7      ,
224         Goto 8      +
225         Goto 9      ABDFGHJKLMNOPQTUVWXYZabdfghjklmnopqtuvwxyz
226         Goto 11     0123456789
227         Goto 14     "
228         Goto 17     /
229         Goto 19     <
230         Goto 22     =
231         Goto 24     >
232         Goto 26     Cc
233         Goto 31     Ee
234         Goto 39     Ii
235         Goto 41     Rr
236         Goto 49     Ss
237
238
239 State 1
240         Goto 1      &09 .. &0D, &20, &85, &A0, &1680, &180E, &2000 ..
    ↪ &200A, &2026, &2028, &2029, &202F, &205F, &3000
241         Accept Whitespace
242
243
244 State 2
245         Accept '-'
246
247
248 State 3
249         Accept '&'
250
251
252 State 4
```

```
253         Accept '('
254
255
256 State 5
257         Accept ')'
258
259
260 State 6
261         Accept '*'
262
263
264 State 7
265         Accept ','
266
267
268 State 8
269         Accept '+'
270
271
272 State 9
273         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
274         Accept Id
275
276
277 State 10
278         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
279         Accept Id
280
281
282 State 11
283         Goto 11      0123456789
284         Goto 12      .
285         Accept NumberLiteral
286
287
288 State 12
289         Goto 13      0123456789
```

```
290
291
292  State 13
293          Goto 13      0123456789
294          Accept NumberLiteral
295
296
297  State 14
298          Goto 15
     ↪   {Space}!#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
299          Goto 16      "
300
301
302  State 15
303          Goto 15
     ↪   {Space}!#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
304          Goto 16      "
305
306
307  State 16
308          Accept StringLiteral
309
310
311  State 17
312          Goto 18      Ee
313          Accept '/'
314
315
316  State 18
317          Accept '/e'
318
319
320  State 19
321          Goto 20      =
322          Goto 21      >
323          Accept '<'
324
325
326  State 20
```

```
327         Accept '<='
328
329
330 State 21
331         Accept '<>'
332
333
334 State 22
335         Goto 23      =
336         Accept '='
337
338
339 State 23
340         Accept '=='
341
342
343 State 24
344         Goto 25      =
345         Accept '>'
346
347
348 State 25
349         Accept '>='
350
351
352 State 26
353         Goto 10
    ↪ 0123456789ABCDEFGHIJKMNOPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
354         Goto 27      Ll
355         Accept Id
356
357
358 State 27
359         Goto 10
    ↪ 0123456789ABCDFGHIJKLMNOPQRSTUVWXYZabcdfghijklmnopqrstuvwxyz
360         Goto 28      Ee
361         Accept Id
362
363
```

```
364 State 28
365         Goto 10
    ↪ 0123456789BCDEFGHIJKLMNOPQRSTUVWXYZbcdefghijklmnopqrstuvwxyz
366         Goto 29      Aa
367         Accept Id
368
369
370 State 29
371         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQSTUVWXYZabcdefghijklmnopqstuvwxyz
372         Goto 30      Rr
373         Accept Id
374
375
376 State 30
377         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
378         Accept clear
379
380
381 State 31
382         Goto 10
    ↪ 0123456789ABCDEFGHIJKMOPQRSTUVWXYZabcdefghijkmopqrstuvwxyz
383         Goto 32      Ll
384         Goto 35      Nn
385         Accept Id
386
387
388 State 32
389         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRTUVWXYZabcdefghijklmnopqrtuvwxyz
390         Goto 33      Ss
391         Accept Id
392
393
394 State 33
395         Goto 10
    ↪ 0123456789ABCDFGHIJKLMNOPQRSTUVWXYZabcdfghijklmnopqrstuvwxyz
396         Goto 34      Ee
```

```
397         Accept Id
398
399
400 State 34
401         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
402         Accept else
403
404
405 State 35
406         Goto 10
    ↪ 0123456789ABCEFGHIJKLMNOPQRSTUVWXYZabcefghijklmnopqrstuvwxyz
407         Goto 36      Dd
408         Accept Id
409
410
411 State 36
412         Goto 10
    ↪ 0123456789ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghjklmnopqrstuvwxyz
413         Goto 37      Ii
414         Accept Id
415
416
417 State 37
418         Goto 10
    ↪ 0123456789ABCDEGHIJKLMNOPQRSTUVWXYZabcdeghijklmnopqrstuvwxyz
419         Goto 38      Ff
420         Accept Id
421
422
423 State 38
424         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
425         Accept endif
426
427
428 State 39
429         Goto 10
    ↪ 0123456789ABCDEGHIJKLMNOPQRSTUVWXYZabcdeghijklmnopqrstuvwxyz
```

```
430         Goto 40      Ff
431         Accept Id
432
433
434 State 40
435         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
436         Accept if
437
438
439 State 41
440         Goto 10
    ↪ 0123456789ABCDFGHIJKLMNOPQRSTUVWXYZabcdfghijklmnopqrstuvwxyz
441         Goto 42      Ee
442         Accept Id
443
444
445 State 42
446         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSUVWXYZabcdefghijklmnopqrsuvwxyz
447         Goto 43      Tt
448         Accept Id
449
450
451 State 43
452         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQSTUVWXYZabcdefghijklmnopqstuvwxyz
453         Goto 44      Rr
454         Accept Id
455
456
457 State 44
458         Goto 10
    ↪ 0123456789ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghjklmnopqrstuvwxyz
459         Goto 45      Ii
460         Accept Id
461
462
463 State 45
```

```
464         Goto 10
    ↪ 0123456789ABCDFGHIJKLMNOPQRSTUVWXYZabcdfghijklmnopqrstuvwxyz
465         Goto 46      Ee
466         Accept Id
467
468
469 State 46
470         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUWXYZabcdefghijklmnopqrstuwxyz
471         Goto 47      Vv
472         Accept Id
473
474
475 State 47
476         Goto 10
    ↪ 0123456789ABCDFGHIJKLMNOPQRSTUVWXYZabcdfghijklmnopqrstuvwxyz
477         Goto 48      Ee
478         Accept Id
479
480
481 State 48
482         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
483         Accept retrieve
484
485
486 State 49
487         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnpqrstuvwxyz
488         Goto 50      Oo
489         Accept Id
490
491
492 State 50
493         Goto 10
    ↪ 0123456789ABCDEFGHIJKLMNOPQSTUVWXYZabcdefghijklmnopqstuvwxyz
494         Goto 51      Rr
495         Accept Id
496
```

```
497
498  State 51
499          Goto 10
     ↪ 0123456789ABCDEFGHIJKLMNOPQRSUVWXYZabcdefghijklmnopqrsuvwxyz
500          Goto 52      Tt
501          Accept Id
502
503
504  State 52
505          Goto 10
     ↪ 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
506          Accept sort
507
508
509
510
511  ========================================
512  LALR States
513  ========================================
514
515  State 0
516          <S'> ::= _ <Statements> (EOF)
     ↪ <S'> ::= ^ <Statements> (EOF)
517          <Statements> ::= _ <Statement> <Statements>
     ↪ <Statements> ::= ^ <Statement> <Statements>
518          <Statements> ::= _ <Statement>
     ↪ <Statements> ::= ^ <Statement>
519          <Statement> ::= _ <BuiltInStatement>
     ↪ <Statement> ::= ^ <BuiltInStatement>
520          <Statement> ::= _ <If>
     ↪ <Statement> ::= ^ <If>
521          <BuiltInStatement> ::= _ <Clear>
     ↪ <BuiltInStatement> ::= ^ <Clear>
522          <BuiltInStatement> ::= _ <Retrieve>
     ↪ <BuiltInStatement> ::= ^ <Retrieve>
523          <BuiltInStatement> ::= _ <Sort>
     ↪ <BuiltInStatement> ::= ^ <Sort>
524          <Clear> ::= _ clear <ClearParam> StringLiteral
     ↪ <Clear> ::= ^ clear <ClearParam> StringLiteral
```

```
525         <Retrieve> ::= _ retrieve <RetrieveParam> StringLiteral
    ↪ <Retrieve> ::= ^ retrieve <RetrieveParam> StringLiteral
526         <Sort> ::= _ sort <RetrieveParam> <SortValue>
    ↪ <Sort> ::= ^ sort <RetrieveParam> <SortValue>
527         <If> ::= _ if '(' Id <Compare> StringLiteral ')' <Statements>
    ↪ <Else> endif        <If> ::= ^ if '(' Id <Compare> StringLiteral ')'
    ↪ <Statements> <Else> endif
528
529         clear s 1
530         if s 2
531         retrieve s 3
532         sort s 4
533         <BuiltInStatement> g 5
534         <Clear> g 6
535         <If> g 7
536         <Retrieve> g 8
537         <Sort> g 9
538         <Statement> g 10
539         <Statements> g 11
540
541
542 State 1
543         Prior States: 0, 10, 34, 36
544
545         <Clear> ::= clear _ <ClearParam> StringLiteral
    ↪ <Clear> ::= clear ^ <ClearParam> StringLiteral
546         <ClearParam> ::= _ '/e'
    ↪ <ClearParam> ::= ^ '/e'
547         <ClearParam> ::= _
    ↪ <ClearParam> ::= ^
548
549         '/e' s 12
550         <ClearParam> g 13
551         StringLiteral r 9
552
553
554 State 2
555         Prior States: 0, 10, 34, 36
556
```

```
557         <If> ::= if _ '(' Id <Compare> StringLiteral ')' <Statements>
    ↪  <Else> endif         <If> ::= if ^ '(' Id <Compare> StringLiteral ')'
    ↪  <Statements> <Else> endif
558
559         '(' s 14
560
561
562 State 3
563         Prior States: 0, 10, 34, 36
564
565         <Retrieve> ::= retrieve _ <RetrieveParam> StringLiteral
    ↪  <Retrieve> ::= retrieve ^ <RetrieveParam> StringLiteral
566         <RetrieveParam> ::= _ '/e'
    ↪  <RetrieveParam> ::= ^ '/e'
567         <RetrieveParam> ::= _
    ↪  <RetrieveParam> ::= ^
568
569         '/e' s 15
570         <RetrieveParam> g 16
571         StringLiteral r 12
572
573
574 State 4
575         Prior States: 0, 10, 34, 36
576
577         <Sort> ::= sort _ <RetrieveParam> <SortValue>
    ↪  <Sort> ::= sort ^ <RetrieveParam> <SortValue>
578         <RetrieveParam> ::= _ '/e'
    ↪  <RetrieveParam> ::= ^ '/e'
579         <RetrieveParam> ::= _
    ↪  <RetrieveParam> ::= ^
580
581         '/e' s 15
582         <RetrieveParam> g 17
583         StringLiteral r 12
584
585
586 State 5
587         Prior States: 0, 10, 34, 36
```

```
588
589         <Statement> ::= <BuiltInStatement> _
    ↪   <Statement> ::= <BuiltInStatement> ^
590
591         (EOF) r 2
592         clear r 2
593         else r 2
594         endif r 2
595         if r 2
596         retrieve r 2
597         sort r 2
598
599
600 State 6
601         Prior States: 0, 10, 34, 36
602
603         <BuiltInStatement> ::= <Clear> _
    ↪   <BuiltInStatement> ::= <Clear> ^
604
605         (EOF) r 4
606         clear r 4
607         else r 4
608         endif r 4
609         if r 4
610         retrieve r 4
611         sort r 4
612
613
614 State 7
615         Prior States: 0, 10, 34, 36
616
617         <Statement> ::= <If> _
    ↪   <Statement> ::= <If> ^
618
619         (EOF) r 3
620         clear r 3
621         else r 3
622         endif r 3
623         if r 3
```

```
624         retrieve r 3
625         sort r 3
626
627
628 State 8
629         Prior States: 0, 10, 34, 36
630
631         <BuiltInStatement> ::= <Retrieve> _
    ↪   <BuiltInStatement> ::= <Retrieve> ^
632
633         (EOF) r 5
634         clear r 5
635         else r 5
636         endif r 5
637         if r 5
638         retrieve r 5
639         sort r 5
640
641
642 State 9
643         Prior States: 0, 10, 34, 36
644
645         <BuiltInStatement> ::= <Sort> _
    ↪   <BuiltInStatement> ::= <Sort> ^
646
647         (EOF) r 6
648         clear r 6
649         else r 6
650         endif r 6
651         if r 6
652         retrieve r 6
653         sort r 6
654
655
656 State 10
657         Prior States: 0, 10, 34, 36
658
659         <Statements> ::= <Statement> _ <Statements>
    ↪   <Statements> ::= <Statement> ^ <Statements>
```

```
660          <Statements> ::= <Statement> _
    ↪  <Statements> ::= <Statement> ^
661          <Statements> ::= _ <Statement> <Statements>
    ↪  <Statements> ::= ^ <Statement> <Statements>
662          <Statements> ::= _ <Statement>
    ↪  <Statements> ::= ^ <Statement>
663          <Statement> ::= _ <BuiltInStatement>
    ↪  <Statement> ::= ^ <BuiltInStatement>
664          <Statement> ::= _ <If>
    ↪  <Statement> ::= ^ <If>
665          <BuiltInStatement> ::= _ <Clear>
    ↪  <BuiltInStatement> ::= ^ <Clear>
666          <BuiltInStatement> ::= _ <Retrieve>
    ↪  <BuiltInStatement> ::= ^ <Retrieve>
667          <BuiltInStatement> ::= _ <Sort>
    ↪  <BuiltInStatement> ::= ^ <Sort>
668          <Clear> ::= _ clear <ClearParam> StringLiteral
    ↪  <Clear> ::= ^ clear <ClearParam> StringLiteral
669          <Retrieve> ::= _ retrieve <RetrieveParam> StringLiteral
    ↪  <Retrieve> ::= ^ retrieve <RetrieveParam> StringLiteral
670          <Sort> ::= _ sort <RetrieveParam> <SortValue>
    ↪  <Sort> ::= ^ sort <RetrieveParam> <SortValue>
671          <If> ::= _ if '(' Id <Compare> StringLiteral ')' <Statements>
    ↪  <Else> endif         <If> ::= ^ if '(' Id <Compare> StringLiteral ')'
    ↪  <Statements> <Else> endif
672
673          clear s 1
674          if s 2
675          retrieve s 3
676          sort s 4
677          <BuiltInStatement> g 5
678          <Clear> g 6
679          <If> g 7
680          <Retrieve> g 8
681          <Sort> g 9
682          <Statement> g 10
683          <Statements> g 18
684          (EOF) r 1
685          else r 1
```

```
686         endif r 1
687
688
689 State 11
690         Prior States: 0
691
692         <S'> ::= <Statements> _ (EOF)
    ↪ <S'> ::= <Statements> ^ (EOF)
693
694         (EOF) a
695
696
697 State 12
698         Prior States: 1
699
700         <ClearParam> ::= '/e' _
    ↪ <ClearParam> ::= '/e' ^
701
702         StringLiteral r 8
703
704
705 State 13
706         Prior States: 1
707
708         <Clear> ::= clear <ClearParam> _ StringLiteral
    ↪ <Clear> ::= clear <ClearParam> ^ StringLiteral
709
710         StringLiteral s 19
711
712
713 State 14
714         Prior States: 2
715
716         <If> ::= if '(' _ Id <Compare> StringLiteral ')' <Statements>
    ↪ <Else> endif        <If> ::= if '(' ^ Id <Compare> StringLiteral ')'
    ↪ <Statements> <Else> endif
717
718         Id s 20
719
```

```
720
721 State 15
722         Prior States: 3, 4
723
724         <RetrieveParam> ::= '/e' _
    ↪   <RetrieveParam> ::= '/e' ^
725
726         StringLiteral r 11
727
728
729 State 16
730         Prior States: 3
731
732         <Retrieve> ::= retrieve <RetrieveParam> _ StringLiteral
    ↪   <Retrieve> ::= retrieve <RetrieveParam> ^ StringLiteral
733
734         StringLiteral s 21
735
736
737 State 17
738         Prior States: 4
739
740         <Sort> ::= sort <RetrieveParam> _ <SortValue>
    ↪   <Sort> ::= sort <RetrieveParam> ^ <SortValue>
741         <SortValue> ::= _ StringLiteral
    ↪   <SortValue> ::= ^ StringLiteral
742         <SortValue> ::= _ StringLiteral ',' <SortValue>
    ↪   <SortValue> ::= ^ StringLiteral ',' <SortValue>
743
744         StringLiteral s 22
745         <SortValue> g 23
746
747
748 State 18
749         Prior States: 10
750
751         <Statements> ::= <Statement> <Statements> _
    ↪   <Statements> ::= <Statement> <Statements> ^
752
```

```
753         (EOF) r 0
754         else r 0
755         endif r 0
756
757
758 State 19
759         Prior States: 13
760
761         <Clear> ::= clear <ClearParam> StringLiteral _
    ↪   <Clear> ::= clear <ClearParam> StringLiteral ^
762
763         (EOF) r 7
764         clear r 7
765         else r 7
766         endif r 7
767         if r 7
768         retrieve r 7
769         sort r 7
770
771
772 State 20
773         Prior States: 14
774
775         <If> ::= if '(' Id _ <Compare> StringLiteral ')' <Statements>
    ↪   <Else> endif        <If> ::= if '(' Id ^ <Compare> StringLiteral ')'
    ↪   <Statements> <Else> endif
776         <Compare> ::= _ '>'
    ↪   <Compare> ::= ^ '>'
777         <Compare> ::= _ '<'
    ↪   <Compare> ::= ^ '<'
778         <Compare> ::= _ '<='
    ↪   <Compare> ::= ^ '<='
779         <Compare> ::= _ '>='
    ↪   <Compare> ::= ^ '>='
780         <Compare> ::= _ '<>'
    ↪   <Compare> ::= ^ '<>'
781         <Compare> ::= _ '='
    ↪   <Compare> ::= ^ '='
782
```

```
783         '<' s 24
784         '<=' s 25
785         '<>' s 26
786         '=' s 27
787         '>' s 28
788         '>=' s 29
789         <Compare> g 30
790
791
792 State 21
793         Prior States: 16
794
795         <Retrieve> ::= retrieve <RetrieveParam> StringLiteral _
    ↪   <Retrieve> ::= retrieve <RetrieveParam> StringLiteral ^
796
797         (EOF) r 10
798         clear r 10
799         else r 10
800         endif r 10
801         if r 10
802         retrieve r 10
803         sort r 10
804
805
806 State 22
807         Prior States: 17, 31
808
809         <SortValue> ::= StringLiteral _
    ↪   <SortValue> ::= StringLiteral ^
810         <SortValue> ::= StringLiteral _ ',' <SortValue>
    ↪   <SortValue> ::= StringLiteral ^ ',' <SortValue>
811
812         ',' s 31
813         (EOF) r 14
814         clear r 14
815         else r 14
816         endif r 14
817         if r 14
818         retrieve r 14
```

```
819         sort r 14
820
821
822 State 23
823         Prior States: 17
824
825         <Sort> ::= sort <RetrieveParam> <SortValue> _
    ↪   <Sort> ::= sort <RetrieveParam> <SortValue> ^
826
827         (EOF) r 13
828         clear r 13
829         else r 13
830         endif r 13
831         if r 13
832         retrieve r 13
833         sort r 13
834
835
836 State 24
837         Prior States: 20
838
839         <Compare> ::= '<' _
    ↪   <Compare> ::= '<' ^
840
841         StringLiteral r 22
842
843
844 State 25
845         Prior States: 20
846
847         <Compare> ::= '<=' _
    ↪   <Compare> ::= '<=' ^
848
849         StringLiteral r 23
850
851
852 State 26
853         Prior States: 20
854
```

```
855         <Compare> ::= '<>' _
    ↪ <Compare> ::= '<>' ^
856
857         StringLiteral r 25
858
859
860 State 27
861         Prior States: 20
862
863         <Compare> ::= '=' _
    ↪ <Compare> ::= '=' ^
864
865         StringLiteral r 26
866
867
868 State 28
869         Prior States: 20
870
871         <Compare> ::= '>' _
    ↪ <Compare> ::= '>' ^
872
873         StringLiteral r 21
874
875
876 State 29
877         Prior States: 20
878
879         <Compare> ::= '>=' _
    ↪ <Compare> ::= '>=' ^
880
881         StringLiteral r 24
882
883
884 State 30
885         Prior States: 20
886
887         <If> ::= if '(' Id <Compare> _ StringLiteral ')' <Statements>
    ↪ <Else> endif        <If> ::= if '(' Id <Compare> ^ StringLiteral ')'
    ↪ <Statements> <Else> endif
```

```
888
889         StringLiteral s 32
890
891
892 State 31
893         Prior States: 22
894
895         <SortValue> ::= StringLiteral ',' _ <SortValue>
    ↪   <SortValue> ::= StringLiteral ',' ^ <SortValue>
896         <SortValue> ::= _ StringLiteral
    ↪   <SortValue> ::= ^ StringLiteral
897         <SortValue> ::= _ StringLiteral ',' <SortValue>
    ↪   <SortValue> ::= ^ StringLiteral ',' <SortValue>
898
899         StringLiteral s 22
900         <SortValue> g 33
901
902
903 State 32
904         Prior States: 30
905
906         <If> ::= if '(' Id <Compare> StringLiteral _ ')' <Statements>
    ↪   <Else> endif        <If> ::= if '(' Id <Compare> StringLiteral ^ ')'
    ↪   <Statements> <Else> endif
907
908         ')' s 34
909
910
911 State 33
912         Prior States: 31
913
914         <SortValue> ::= StringLiteral ',' <SortValue> _
    ↪   <SortValue> ::= StringLiteral ',' <SortValue> ^
915
916         (EOF) r 15
917         clear r 15
918         else r 15
919         endif r 15
920         if r 15
```

```
921         retrieve r 15
922         sort r 15
923
924
925 State 34
926         Prior States: 32
927
928         <If> ::= if '(' Id <Compare> StringLiteral ')' _ <Statements>
    ↪   <Else> endif        <If> ::= if '(' Id <Compare> StringLiteral ')' ^
    ↪   <Statements> <Else> endif
929         <Statements> ::= _ <Statement> <Statements>
    ↪   <Statements> ::= ^ <Statement> <Statements>
930         <Statements> ::= _ <Statement>
    ↪   <Statements> ::= ^ <Statement>
931         <Statement> ::= _ <BuiltInStatement>
    ↪   <Statement> ::= ^ <BuiltInStatement>
932         <Statement> ::= _ <If>
    ↪   <Statement> ::= ^ <If>
933         <BuiltInStatement> ::= _ <Clear>
    ↪   <BuiltInStatement> ::= ^ <Clear>
934         <BuiltInStatement> ::= _ <Retrieve>
    ↪   <BuiltInStatement> ::= ^ <Retrieve>
935         <BuiltInStatement> ::= _ <Sort>
    ↪   <BuiltInStatement> ::= ^ <Sort>
936         <Clear> ::= _ clear <ClearParam> StringLiteral
    ↪   <Clear> ::= ^ clear <ClearParam> StringLiteral
937         <Retrieve> ::= _ retrieve <RetrieveParam> StringLiteral
    ↪   <Retrieve> ::= ^ retrieve <RetrieveParam> StringLiteral
938         <Sort> ::= _ sort <RetrieveParam> <SortValue>
    ↪   <Sort> ::= ^ sort <RetrieveParam> <SortValue>
939         <If> ::= _ if '(' Id <Compare> StringLiteral ')' <Statements>
    ↪   <Else> endif        <If> ::= ^ if '(' Id <Compare> StringLiteral ')'
    ↪   <Statements> <Else> endif
940
941         clear s 1
942         if s 2
943         retrieve s 3
944         sort s 4
945         <BuiltInStatement> g 5
```

```
946         <Clear> g 6
947         <If> g 7
948         <Retrieve> g 8
949         <Sort> g 9
950         <Statement> g 10
951         <Statements> g 35
952
953
954 State 35
955         Prior States: 34
956
957         <If> ::= if '(' Id <Compare> StringLiteral ')' <Statements> _
    ↪   <Else> endif        <If> ::= if '(' Id <Compare> StringLiteral ')'
    ↪   <Statements> ^ <Else> endif
958         <Else> ::= _ else <Statements>
    ↪   <Else> ::= ^ else <Statements>
959         <Else> ::= _
    ↪   <Else> ::= ^
960
961         else s 36
962         <Else> g 37
963         endif r 20
964
965
966 State 36
967         Prior States: 35
968
969         <Else> ::= else _ <Statements>
    ↪   <Else> ::= else ^ <Statements>
970         <Statements> ::= _ <Statement> <Statements>
    ↪   <Statements> ::= ^ <Statement> <Statements>
971         <Statements> ::= _ <Statement>
    ↪   <Statements> ::= ^ <Statement>
972         <Statement> ::= _ <BuiltInStatement>
    ↪   <Statement> ::= ^ <BuiltInStatement>
973         <Statement> ::= _ <If>
    ↪   <Statement> ::= ^ <If>
974         <BuiltInStatement> ::= _ <Clear>
    ↪   <BuiltInStatement> ::= ^ <Clear>
```

```
975         <BuiltInStatement> ::= _ <Retrieve>
    ↪  <BuiltInStatement> ::= ^ <Retrieve>
976         <BuiltInStatement> ::= _ <Sort>
    ↪  <BuiltInStatement> ::= ^ <Sort>
977         <Clear> ::= _ clear <ClearParam> StringLiteral
    ↪  <Clear> ::= ^ clear <ClearParam> StringLiteral
978         <Retrieve> ::= _ retrieve <RetrieveParam> StringLiteral
    ↪  <Retrieve> ::= ^ retrieve <RetrieveParam> StringLiteral
979         <Sort> ::= _ sort <RetrieveParam> <SortValue>
    ↪  <Sort> ::= ^ sort <RetrieveParam> <SortValue>
980         <If> ::= _ if '(' Id <Compare> StringLiteral ')' <Statements>
    ↪  <Else> endif        <If> ::= ^ if '(' Id <Compare> StringLiteral ')'
    ↪  <Statements> <Else> endif
981
982         clear s 1
983         if s 2
984         retrieve s 3
985         sort s 4
986         <BuiltInStatement> g 5
987         <Clear> g 6
988         <If> g 7
989         <Retrieve> g 8
990         <Sort> g 9
991         <Statement> g 10
992         <Statements> g 38
993
994
995  State 37
996         Prior States: 35
997
998         <If> ::= if '(' Id <Compare> StringLiteral ')' <Statements> <Else>
    ↪  _ endif        <If> ::= if '(' Id <Compare> StringLiteral ')'
    ↪  <Statements> <Else> ^ endif
999
1000        endif s 39
1001
1002
1003 State 38
1004        Prior States: 36
```

```
1005
1006        <Else> ::= else <Statements> _
    ↪  <Else> ::= else <Statements> ^
1007
1008        endif r 19
1009
1010
1011  State 39
1012        Prior States: 37
1013
1014        <If> ::= if '(' Id <Compare> StringLiteral ')' <Statements> <Else>
    ↪  endif _        <If> ::= if '(' Id <Compare> StringLiteral ')'
    ↪  <Statements> <Else> endif ^
1015
1016        (EOF) r 18
1017        clear r 18
1018        else r 18
1019        endif r 18
1020        if r 18
1021        retrieve r 18
1022        sort r 18
```

Appendix E

Generating Abstract Syntax Tree Using ANTLR for Other 4GLs

This appendix illustrates the code samples used to extend the author's method over Informix and Apex. This appendix presents the ASTs for both languages.

In addition to the AST, the lexer and 'Visitor' set of classes, Parr [188], for each rule has been generated and implemented similar to the work that was done in appendix C. Nodes are traversed using the 'visitor' design pattern, Visser [344].

The figures in this appendix were processed to extract chunking trees that were used for comparison with the output (Informix 10.4.2, and Apex 10.4.3).

E.1 Informix

E.1.1 Informix Sample Code Snippet

The sample code transformed for Informix include the following snippet. This code performs a loop and defines a variable of the data-type number.

```
MAIN
DEFINE i SMALLINT
FOR i = 1 TO 10
   DISPLAY i
END FOR
END MAIN
```

E.1.2 Informix Abstract Syntax Tree (AST)

The snippet has been represent by the AST in figure E.1.

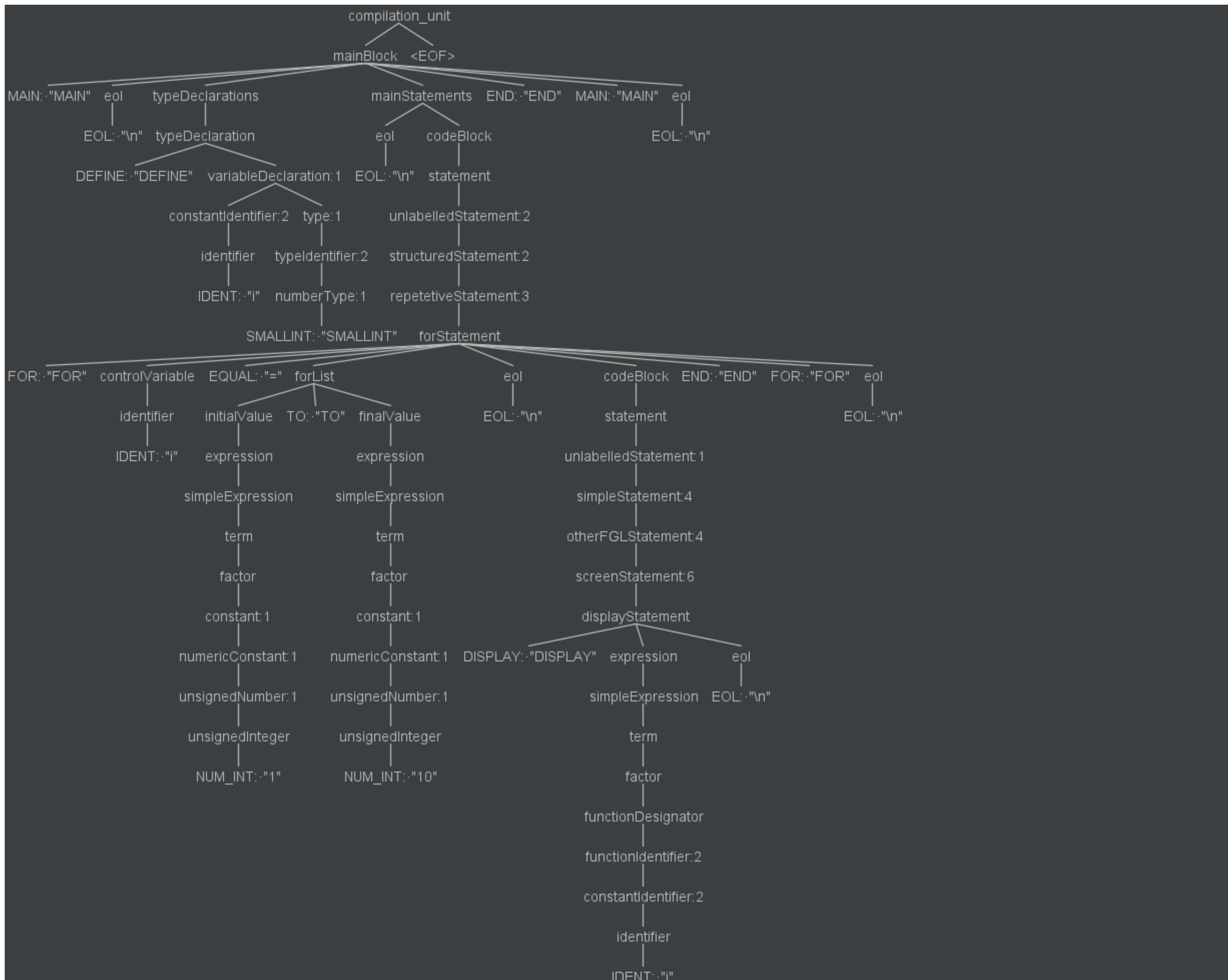

Figure E.1: Informix AST for Sample Code

E.1.3 Informix Snippet into C# Output

The code above has been translated to the following C# code:

```
public void Main()
{
    int16 i;
    for(i=1;i<=10;i++)
```

```
    {
        Display(i);
    }
}
```

E.2 Apex

E.2.1 APEX Sample Code Snippet

The sample code transformed for Apex include the following snippet.

```
@IsTest
private class TestRunAs {
    public static testMethod void testRunAs() {
        String debugginValue = "Runtime started OK";
        System.debug("Runtime status" + debugginValue);
    }
}
```

E.2.2 APEX Abstract Syntax Tree (AST)

The above sample code has been represented by the following AST in figure E.2

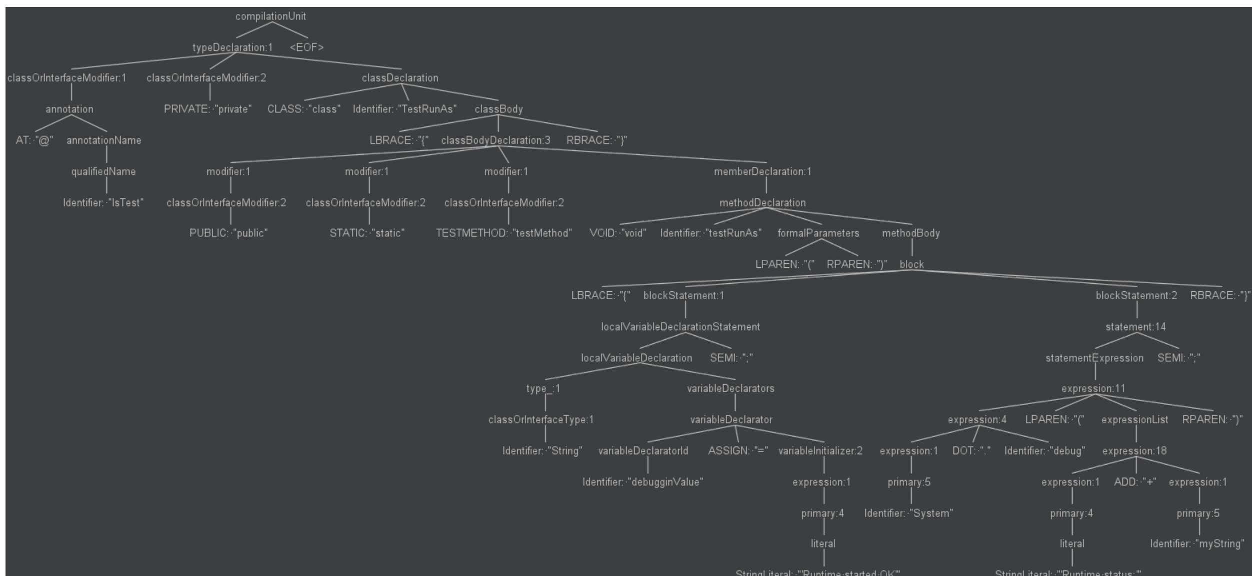

Figure E.2: AST for Apex Sample Code

E.2.3 APEX Snippet into C# Output

The code above has been translated to the following C# code:

```
[IsTest]
private class TestRunAs
{
    public static void testRunAs()
    {
        string debugginValue = "Runtime started OK";
        Debug.Write("Runtime status" + debugginValue);
    }
}
```

and this C# code has been represented by the AST in figure E.3 below

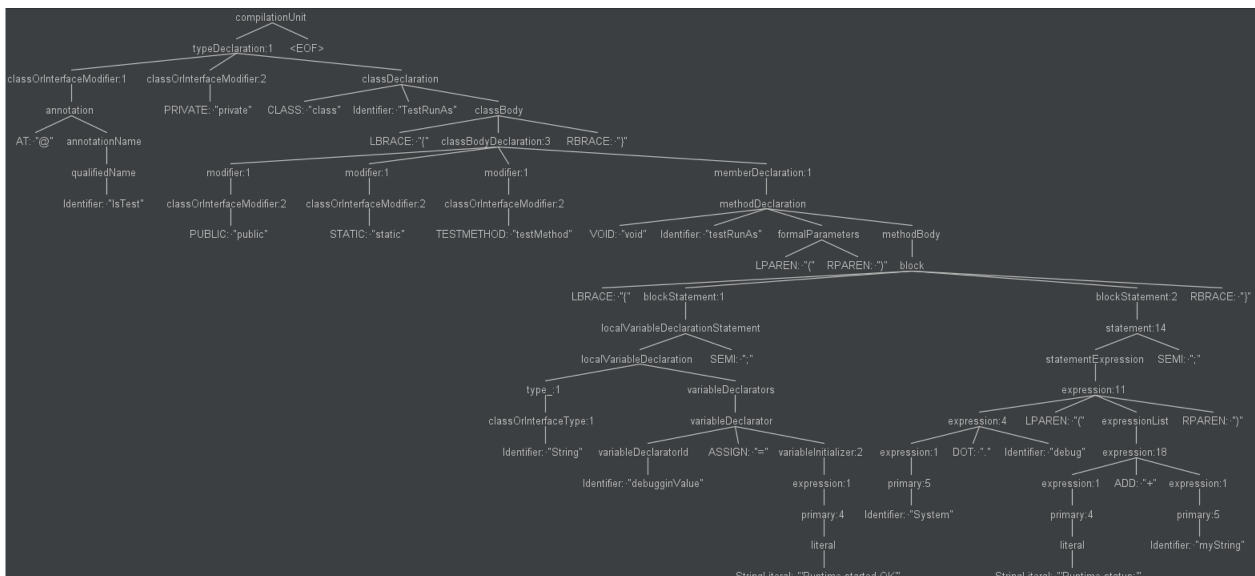

Figure E.3: AST for Apex Sample Code Into C#

Appendix F

Programming Languages Timeline Between 1950 and 2003

Year	Name
FORTRAN Nov.	1954
FORTRAN Oct.	1956
FORTRAN II 1957	1957
FORTRAN III	1958
Flow-Matic	1958
ALGOL 58	1958
USP	1958
COBOL	1959
LISP 1	1959
APL	1960
ALGOL 60	1960
COBOL 61	1961
FORTRAN IV	1962
Extended	1962
usp 1.5	1962
SNOBOL	1962
CPL	1963
PL/I	1964
Simula I	1964
BASIC May 1,	1964
SNOBOL 2 April	1964
SNOBOL 3	1965

Year	Name
FORTRAN (FORTRAN 66 ANS)	1966
MUMPS	1966
ISWIM	1966
BCPL	1967
Simula 67	1967
SNOBOL 4	1967
COBOL 68 ANS	1968
ALGOL 68 Dec.	1968
Forth	1969
Smalltalk	1969
B	1969
Prolog	1970
Pascal	1970
SH	1970
c	1971
PLIM	1972
Smalltalk-72	1972
ML	1973
ANSI	1974
CLU	1974
Modula	1975
Modula	1975
MS Basic 2.0	1975
Scheme	1975
PL1/ANS	1976
Smalltalk-76	1976
Smalltalk-76	1976
SASL	1976
MUMPS (ANSI) sept. 15,	1977
Mesa	1977
Mesa	1977
FORTRAN V FORTRAN 77 April	1978
Smalltalk	1978
awk	1978
Scheme MIT	1978
Rex 1.00 May	1979
Modula 2	1979
Ada	1979

Year	Name
Modula 2	1979
Rex 2.00	1980
C with Classes April	1980
Smalltalk-80	1980
C with Classes April	1980
smalltalk-80	1980
KRC	1981
Rex 3.00	1982
PostScript	1982
Prolog II Oct.	1982
Miranda	1982
Sharp APL	1983
Pascal AFNOR	1983
Ada 83 ANSI Jan.	1983
Objective-C 1983 Jut'/	1983
Cedar	1983
Ada 83 ANSI Jan.	1983
Objective-C	1983
C++	1983
Cedar	1983
Rexx 3.20	1984
Prolog III	1984
APL2 August	1984
Concurrent C	1984
Concurrent	1984
Common Lisp	1984
Scheme 84	1984
SML	1984
COBOL 85 OSI/ANSI	1985
Object Pascal	1985
Object Pascal	1985
nawk	1985
Object Logo	1986
MUMPS (FIPS)	1986
Eiffel	1986
OO Forth	1987
Oberon	1987
ABC	1987

Year	Name
Ada ISO	1987
Smaltalk 78	1987
Ada ISO	1987
pert 1.000 Dec. 18,	1987
Haskell 1.0	1987
Caml	1987
A	1988
MOdula 3	1988
Tcl/Tk end	1988
Tcl mid	1988
Tcl/Tk end	1988
MOdula 3	1988
ANSI C (C89)	1989
ANSI C (C89)	1989
perl 3.000 oct. 18,	1989
bash	1989
Clos	1989
Borland Pascal Object ANSI C (C89)	1989
perl 3.000 oct. 18,	1989
bash	1989
Clos	1989
ISO C	1990
ISO C (C90) Dec. 15,	1990
ISO c (C90) Dec. 15,	1990
Scheme IEEE	1990
Haskell 1.1 April l,	1990
SML '90	1990
ISO c (C90) Dec. 15,	1990
Scheme IEEE	1990
Haskell 1.1 April l,	1990
SML '90	1990
Oberon-2	1991
Fortran 90 ISO	1991
Python	1991
Oak June	1991
NetRexx	1991
NetRexx	1991
oak June	1991

Year	Name
Sather 0.1 June	1991
Perl 4.000 March 21,	1991
Visual Basic 1.0 May 20,	1991
caml 2-6.1	1991
Oberon-2	1991
Fortran 90 ISO	1991
Python	1991
NetRexx	1991
oak June	1991
Sather 0.1 June	1991
Perl 4.000 March 21,	1991
Visual Basic 1.0 May 20,	1991
Caml 2-6.1	1991
PostScript level 2	1992
A+	1992
MUMPS ISO	1992
Cmm	1992
Cmm	1992
ksh	1992
Dylan	1992
Haskell 1.2 March	1992
PostScript level 2	1992
MUMPS ISO	1992
Cmm	1992
ksh	1992
Dylan	1992
Haskell 1.2 March	1992
Ruby Feb. 24,	1993
AppleScript	1993
Ruby Feb. 24,	1993
AppleScript	1993
Caml 3.1 1993	1993
M	1994
Sather 1.0	1994
Perl 5.000 oct. 18,	1994
Common Lisp ANSI Dec.8,	1994
Smalltalk-74	1994
M ANSI Dec.8,	1995

Year	Name
Open M Dec.11,	1995
Delphi March 2,	1995
Ada 95	1995
LiveScript	1995
Java 1 May 23,	1995
Ruby 0.95 Dec.	1995
JavaScript Dec.	1995
self 4.0 July 10,	1995
PHP/FI	1995
Sather 1.1	1995
Sather 1.1 sept.	1995
VBScript	1995
PostScript level 3 sept. 11	1996
K	1996
Iso c (C95) April l,	1996
JScript May	1996
Eiffel 4 Dec. 11,	1996
Objective Caml	1996
Haskell 1.3 May	1996
Object Rexx Feb. 25,	1997
Fortran 95 ISO Dec. 15,	1997
Prolog IV	1997
OO COBOL	1997
Modula 2 ISO	1997
ECMAScript June	1997
Ruby 1.1 alpha O AL ust 13,	1997
PHP 2.0 NOV. 13,	1997
Haskell 1.4 April	1997
SML '97	1997
C++ ANSI/ISO	1998
Eiffel 4.2 Feb. 6,	1998
PHP 3.0 June 6,	1998
Perl 5.005_50 July 26,	1998
Visual Basic 6.0 June 16,	1998
Scheme R5RS	1998
TCI/Tk 8.1 APRIL	1999
TC/Tk 8.2.3 Dec. 16,	1999
M ISO	1999

Year	Name
Delphi 5 August,	1999
Python 1.5.2 April 13,	1999
NetRexx 1.150 July 23,	1999
ECMAScript ed3 Dec.,	1999
Ruby 1.3.2 April ,	1999
Sather 1.2.1 Nov. 4,	1999
Haskell 98 Feb.	1999
ISO C (C99) Python 1.6 sept 5,	2000
Python 2.0 Oct. 16,	2000
ActionScript July	2000
Java 2 v1.3 May 8,	2000
Ruby 1.6.1 sept. 27,	2000
PHP 4.0 May 22,	2000
Perl 5.6.O March 28,	2000
Perl 5.7,0 sept 2,	2000
VB.NET	2000
TCI/Tk 8.3 oct. 22,	2001
Delphi 6 May 1,	2001
Python 2.1 April 17,	2001
COBOL 2002 (draft)	2001
C# (ECMA) Dec. 13,	2001
Ruby 1.6.5 sept. 19,	2001
Python 2.2 Dec. 21,	2001
Self 4.1 August 7,	2001
PHP 4.1.0 Dec. 8,	2001
TCI/Tk 8.4 sept. 10,	2002
Fortran 2000 (draft) sept. 30,	2002
Delphi 7 August 6,	2002
Python 2.2.1 April 10,	2002
Java 2 early access Feb. 6,	2002
RUBY 1.6.7 March 1,	2002
Python 2.2.2 Oct. 14,	2002
Java 2 early access Feb. 6,	2002
Java 2 (vl .4.0_01) June 4,	2002
June 4,	2002
Ruby 1.6.8 Dec. 24,	2002
JAVA (1.4.1) SEP	2002
self 4.1.6 sept.	2002

Year	Name
PHP 4.2.O April 22,	2002
PHP 4.2.2 July 22,	2002
PHP 4.2.3 sept. 6,	2002
PHP 4.3.O Dec. 27,	2002
TCI/Tk 8.4.2 March 3,	2003
Python 2.3a2 Feb. 19,	2003
TCL/Tk 8.4.3 May 20,	2003
Python 2.2.3 MAY	2003
Python 2.3 JULY	2003
Python 2.3.1 SEPT	2003
ActionScript 2.0 SEPT	2003
Java 2 (v1.5) early release Dec. 19,	2003
Java 2 (1.4.1_2) June 11,	2003
PHP 4.3.1 Feb. 17,	2003
PHP 4.3.2 May 29,	2003
PHP 4.3.3 August 25,	2003
Perl 5.8.2 Oct,	2003

Source: Lévénez [1]

Appendix G

Uniface TABLE Element

```
1  <TABLE>
2  <DSC name="UFORMAT" model="DICT" system="S" pseudo ="73" level="1" noupdate="0"
3   rbk="0" ffsql="0" transnr="0" segsize="0" ufocc="0" charset=".U">
4  <FLD name="ULABEL" seqno="1" type="S" level="2" pack="0" scale="0" length="16"
5   pointer="0" inum="1" ufocc="0" mandatory="yes" idxnum="1" idxsnr="101" />
6  <FLD name="UVAR" seqno="2" type="S" level="2" pack="0" scale="0" length="16"
7   pointer="0" inum="1" ufocc="0" mandatory="yes" idxnum="1" idxsnr="102" />
8  <FLD name="ULAN" seqno="3" type="S" level="2" pack="0" scale="0" length="3"
9   pointer="0" inum="1" ufocc="0" mandatory="yes" idxnum="1" idxsnr="103" />
10 <FLD name="UTIMESTAMP" seqno="4" type="E" level="2" pack="0" scale="0"
   ↪   length="15"
11  pointer="0" inum="0" ufocc="0" />
12 <FLD name="UDESCR" seqno="5" type="S" level="2" pack="0" scale="0" length="25"
13  pointer="0" inum="0" ufocc="0" />
14 <FLD name="UDEFAULT" seqno="6" type="S" level="2" pack="0" scale="0" length="1"
15  pointer="0" inum="0" ufocc="0" />
16 <FLD name="UCOMMENT" seqno="7" type="S" level="2" pack="128" scale="0"
   ↪   length="0"
17  pointer="0" inum="0" ufocc="0" varinfo=",0,0,0,,1,0,1,\1D,0,0,0,," />
18 <FLD name="UPOPMVE" seqno="8" type="S" level="2" pack="141" scale="0"
   ↪   length="0"
19  pointer="0" inum="0" ufocc="0" varinfo=",1,0,2,\1F\C2,0,0,0,,0,0,0,," />
20 <FLD name="UPOPCPY" seqno="9" type="S" level="2" pack="141" scale="0"
   ↪   length="0"
21  pointer="0" inum="0" ufocc="0" varinfo=",1,0,2,\1F\C3,0,0,0,,0,0,0,," />
```

```
22 <FLD name="UPOPLNK" seqno="10" type="S" level="2" pack="141" scale="0"
   ↪ length="0"
23  pointer="0" inum="0" ufocc="0" varinfo=",1,0,2,\1F\C4,0,0,0,,0,0,0,," />
24 <FLD name="UPOPCAN" seqno="11" type="S" level="2" pack="141" scale="0"
   ↪ length="0"
25  pointer="0" inum="0" ufocc="0" varinfo=",1,0,2,\1F\C5,0,0,0,,0,0,0,," />
26 <FLD name="UGLYPH" seqno="12" type="S" level="2" pack="141" scale="0"
   ↪ length="0"
27  pointer="0" inum="0" ufocc="0" varinfo=",1,0,2,\1F\C6,0,0,0,,0,0,0,," />
28 </DSC>
29 <OCC>
30 <DAT name="ULABEL">DUMMY_FORMAT</DAT>
31 <DAT name="UVAR">GENLIB</DAT>
32 <DAT name="ULAN">USA</DAT>
33 <DAT name="UTIMESTAMP">2005-08-03T14:21:02.00</DAT>
34 <DAT name="UDEFAULT">2</DAT>
35 <DAT name="UPOPMVE" xml:space='preserve'>Move Here</DAT>
36 <DAT name="UPOPCPY" xml:space='preserve'>Copy Here</DAT>
37 <DAT name="UPOPLNK" xml:space='preserve'>Create Shortcut(s) Here</DAT>
38 <DAT name="UPOPCAN">Cancel</DAT>
39 </OCC>
40 </TABLE>
```

Appendix H

C# Code to Transpile Mini-4GL into C#

```
1  using System;
2  using System.IO;
3  using System.Runtime.Serialization;
4  using com.calitha.goldparser.lalr;
5  using com.calitha.commons;
6
7  namespace com.calitha.goldparser
8  {
9
10     [Serializable()]
11     public class SymbolException : System.Exception
12     {
13         public SymbolException(string message) : base(message)
14         {
15         }
16
17         public SymbolException(string message,
18             Exception inner) : base(message, inner)
19         {
20         }
21
22         protected SymbolException(SerializationInfo info,
23             StreamingContext context) : base(info, context)
```

```
24          {
25          }
26
27      }
28
29      [Serializable()]
30      public class RuleException : System.Exception
31      {
32
33          public RuleException(string message) : base(message)
34          {
35          }
36
37          public RuleException(string message,
38                               Exception inner) : base(message, inner)
39          {
40          }
41
42          protected RuleException(SerializationInfo info,
43                                  StreamingContext context) : base(info, context)
44          {
45          }
46
47      }
48
49      enum SymbolConstants : int
50      {
51          SYMBOL_EOF                = 0, // (EOF)
52          SYMBOL_ERROR              = 1, // (Error)
53          SYMBOL_WHITESPACE         = 2, // Whitespace
54          SYMBOL_BY                 = 3, // By
55          SYMBOL_CODECOLONNUM       = 4, // 'Code: #'
56          SYMBOL_DESCRIPTIONCOLON = 5, // 'Description: '
57          SYMBOL_IDS                = 6, // Ids
58          SYMBOL_ITEM               = 7, // Item
59          SYMBOL_STRINGLITERAL      = 8, // StringLiteral
60          SYMBOL_STRINGLITERALS     = 9, // StringLiterals
61          SYMBOL_CODE               = 10, // <Code>
62          SYMBOL_COMPONENTS         = 11, // <Components>
```

```
63        SYMBOL_DESCRIPTION       = 12, // <Description>
64        SYMBOL_MATERIAL          = 13, // <Material>
65        SYMBOL_MATERIALS         = 14, // <Materials>
66        SYMBOL_OP                = 15, // <OP>
67        SYMBOL_PROGRAM           = 16  // <Program>
68    };
69
70    enum RuleConstants : int
71    {
72        RULE_PROGRAM                                    =  0, // <Program> ::=
          ↪   <Code> <Description> <Components>
73        RULE_CODE_CODECOLONNUM                          =  1, // <Code> ::=
          ↪   'Code: #' <OP>
74        RULE_OP_STRINGLITERAL_IDS                       =  2, // <OP> ::=
          ↪   StringLiteral Ids
75        RULE_DESCRIPTION_DESCRIPTIONCOLON_STRINGLITERAL =  3, // <Description>
          ↪   ::= 'Description: ' StringLiteral
76        RULE_COMPONENTS                                 =  4, // <Components>
          ↪   ::= <Materials>
77        RULE_MATERIALS_ITEM_BY_IDS_ITEM_BY_IDS          =  5, // <Materials>
          ↪   ::= Item By Ids Item By Ids
78        RULE_MATERIALS                                  =  6, // <Materials>
          ↪   ::= <Material>
79        RULE_MATERIAL_ITEM_BY_IDS                       =  7  // <Material> ::=
          ↪   Item By Ids
80    };
81
82    public class MyParser
83    {
84        private LALRParser parser;
85
86        public MyParser(string filename)
87        {
88            FileStream stream = new FileStream(filename,
89                                               FileMode.Open,
90                                               FileAccess.Read,
91                                               FileShare.Read);
92            Init(stream);
93            stream.Close();
```

```
94          }
95
96          public MyParser(string baseName, string resourceName)
97          {
98              byte[] buffer = ResourceUtil.GetByteArrayResource(
99                  System.Reflection.Assembly.GetExecutingAssembly(),
100                 baseName,
101                 resourceName);
102             MemoryStream stream = new MemoryStream(buffer);
103             Init(stream);
104             stream.Close();
105         }
106
107         public MyParser(Stream stream)
108         {
109             Init(stream);
110         }
111
112         private void Init(Stream stream)
113         {
114             CGTReader reader = new CGTReader(stream);
115             parser = reader.CreateNewParser();
116             parser.TrimReductions = false;
117             parser.StoreTokens = LALRParser.StoreTokensMode.NoUserObject;
118
119             parser.OnTokenError += new
                ↪ LALRParser.TokenErrorHandler(TokenErrorEvent);
120             parser.OnParseError += new
                ↪ LALRParser.ParseErrorHandler(ParseErrorEvent);
121         }
122
123         public void Parse(string source)
124         {
125             NonterminalToken token = parser.Parse(source);
126             if (token != null)
127             {
128                 Object obj = CreateObject(token);
129
130             }
```

```
131         }
132
133         private Object CreateObject(Token token)
134         {
135             if (token is TerminalToken)
136                 return CreateObjectFromTerminal((TerminalToken)token);
137             else
138                 return CreateObjectFromNonterminal((NonterminalToken)token);
139         }
140
141         private Object CreateObjectFromTerminal(TerminalToken token)
142         {
143             switch (token.Symbol.Id)
144             {
145                 case (int)SymbolConstants.SYMBOL_EOF :
146                 //(EOF)
147
148                 return EOF;
149
150                 case (int)SymbolConstants.SYMBOL_ERROR :
151                 //(Error)
152
153                 return Error;
154
155                 case (int)SymbolConstants.SYMBOL_WHITESPACE :
156                 //Whitespace
157
158                 return Whitespace;
159
160                 case (int)SymbolConstants.SYMBOL_BY :
161                 //By
162
163                 return By;
164
165                 case (int)SymbolConstants.SYMBOL_CODECOLONNUM :
166                 //'Code: #'
167
168                 return Code;
169
```

```
170                 case (int)SymbolConstants.SYMBOL_DESCRIPTIONCOLON :
171                 //'Description: '
172
173                 return Description;
174
175                 case (int)SymbolConstants.SYMBOL_IDS :
176                 //Ids
177
178                 return Ids;
179
180                 case (int)SymbolConstants.SYMBOL_ITEM :
181                 //Item
182
183                 return Item;
184
185                 case (int)SymbolConstants.SYMBOL_STRINGLITERAL :
186                 //StringLiteral
187
188                 return StringLiteral;
189
190                 case (int)SymbolConstants.SYMBOL_STRINGLITERALS :
191                 //StringLiterals
192
193                 return StringLiterals;
194
195                 case (int)SymbolConstants.SYMBOL_CODE :
196                 //<Code>
197
198                 return Code;
199
200                 case (int)SymbolConstants.SYMBOL_COMPONENTS :
201                 //<Components>
202
203                 return Components;
204
205                 case (int)SymbolConstants.SYMBOL_DESCRIPTION :
206                 //<Description>
207
208                 return Description;
```

```
209
210                 case (int)SymbolConstants.SYMBOL_MATERIAL :
211                 //<Material>
212
213                 return Material;
214
215                 case (int)SymbolConstants.SYMBOL_MATERIALS :
216                 //<Materials>
217
218                 return Materials;
219
220                 case (int)SymbolConstants.SYMBOL_OP :
221                 //<OP>
222
223                 return OP;
224
225                 case (int)SymbolConstants.SYMBOL_PROGRAM :
226                 //<Program>
227
228                 return Program;
229
230             }
231             throw new SymbolException("Unknown symbol");
232         }
233
234         public Object CreateObjectFromNonterminal(NonterminalToken token)
235         {
236             switch (token.Rule.Id)
237             {
238                 case (int)RuleConstants.RULE_PROGRAM :
239                 //<Program> ::= <Code> <Description> <Components>
240
241                 return RULE_PROGRAM;
242
243                 case (int)RuleConstants.RULE_CODE_CODECOLONNUM :
244                 //<Code> ::= 'Code: #' <OP>
245
246                 return RULE_CODE_CODECOLONNUM;
247
```

```
248                 case (int)RuleConstants.RULE_OP_STRINGLITERAL_IDS :
249                 //<OP> ::= StringLiteral Ids
250
251                 return RULE_OP_STRINGLITERAL_IDS;
252
253                 case
                     ↪  (int)RuleConstants.RULE_DESCRIPTION_DESCRIPTIONCOLON_STRINGLITERAL
                     ↪  :
254                 //<Description> ::= 'Description: ' StringLiteral
255
256                 return RULE_DESCRIPTION_DESCRIPTIONCOLON_STRINGLITERAL;
257
258                 case (int)RuleConstants.RULE_COMPONENTS :
259                 //<Components> ::= <Materials>
260
261                 return RULE_COMPONENTS;
262
263                 case (int)RuleConstants.RULE_MATERIALS_ITEM_BY_IDS_ITEM_BY_IDS
                     ↪  :
264                 //<Materials> ::= Item By Ids Item By Ids
265
266                 return RULE_MATERIALS_ITEM_BY_IDS_ITEM_BY_IDS;
267
268                 case (int)RuleConstants.RULE_MATERIALS :
269                 //<Materials> ::= <Material>
270
271                 return RULE_MATERIALS;
272
273                 case (int)RuleConstants.RULE_MATERIAL_ITEM_BY_IDS :
274                 //<Material> ::= Item By Ids
275
276                 return RULE_MATERIAL_ITEM_BY_IDS;
277
278             }
279             throw new RuleException("Unknown rule");
280         }
281
282         private void TokenErrorEvent(LALRParser parser, TokenErrorEventArgs
             ↪  args)
```

```
283         {
284             string message = "Token error with input:
                ↪   '"+args.Token.ToString()+"'";
285
286         }
287
288         private void ParseErrorEvent(LALRParser parser, ParseErrorEventArgs
            ↪   args)
289         {
290             string message = "Parse error caused by token:
                ↪   '"+args.UnexpectedToken.ToString()+"'";
291
292         }
293
294     }
295 }
```

Appendix I

Using String Template Methods to Implement Template Meta-Programming

The code below shows the combination of using ANTLR with String Templates to generate code using template meta-programming methods.

```
import org.antlr.runtime.ANTLRReaderStream;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import org.stringtemplate.v4.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.File;
import java.io.FileOutputStream;
import java.io.Writer;
import java.lang.ArrayIndexOutOfBoundsException;
import java.util.ArrayList;
```

```
19  import java.util.HashMap;
20  import java.util.Map;
21
22  public class ElementRunner {
23    private static final String fileName = "snippet.txt";
24    private static final String outputName = "output.txt";
25    private static File file;
26    private static ArrayList < String > stringList = new ArrayList < String > ();
27    public static void main(String[] args) throws Exception {
28
29      //read from source code text
30      // This will reference one line at a time
31      String line = null;
32      String content = "";
33      FileOutputStream fop = null;
34      BufferedWriter bufferedWriter = null;
35      BufferedWriter bw = null;
36      FileWriter fw = null;
37
38      file = new File("output.txt");
39      fop = new FileOutputStream(file);
40
41      try {
42        FileReader fileReader =
43          new FileReader(fileName);
44        BufferedReader bufferedReader =
45          new BufferedReader(fileReader);
46
47        while ((line = bufferedReader.readLine()) != null) {
48
49          if (line.trim().startsWith(";")) {
50            line = line.trim();
51            int len = line.length();
52            UnifaceParser parser = getParserTree("//");
53            ParseTree tree = parser.comment();
54            ST element = new ST("<comment>" + line.substring(1));
55            element.add("comment", tree.getText());
56            stringList.add(element.render());
57          } else {
```

```
58  if (line.trim().startsWith("if")) {
59    content = "";
60    line = line.trim();
61    if (line.indexOf("=") > 0) {
62      if (line.indexOf(".") < 0) {
63        if (line.indexOf("&amp;") < 0) {
64          String[] str = line.split("=");
65          if (line.trim().indexOf("!") < 0) {
66            content += str[0] + "==" + str[1] + "{";
67          } else {
68            content += str[0] + "=" + str[1] + "{";
69          }
70          stringList.add(content);
71
72        } else {
73          String[] str = line.split("&amp;");
74          content = str[0] + "&&" + str[1];
75          stringList.add(content);
76        }
77
78      } else {
79        line = line.trim();
80        String[] str = line.split("=");
81        String str1 = str[0].substring(4);
82        if (str1.indexOf(".") > 0) {
83          String[] str2 = str1.split("\\.");
84          ST h = new
           ↪  ST("if(<classname>.<methodname>==<suffixcondition>{");
85          h.add("classname", str2[1].trim());
86          h.add("methodname", str2[0]);
87          h.add("suffixcondition", str[1]);
88          content = h.render();
89        }
90
91        stringList.add(content);
92
93      }
94
95    }
```

```
96
97              } else if (line.trim().startsWith("until")) {
98                if (line.trim().indexOf("&lt;=") > 0) {
99                  String[] str = line.trim().split("&lt;=");
100                 UnifaceParser parser = getParserTree(str[0] + "," + str[1]);
101                 ParseTree tree = parser.untill();
102                 ST h = new ST("<param1>>=<param2>{");
103                 h.add("param1", str[0]);
104                 h.add("param2", str[1]);
105                 stringList.add(h.render());
106               }
107             } else if (line.trim().startsWith("elseif")) {
108               content = "";
109               line = line.trim();
110               if (line.indexOf("=") > 0) {
111                 if (line.indexOf(".") < 0) {
112                   if (line.indexOf("&amp;") < 0) {
113                     String[] str = line.split("=");
114                     if (line.trim().indexOf("!") < 0) {
115                       ST element = new ST("else if(<param1>==<param2>{)");
116                       element.add("param1", str[0]);
117                       element.add("param2", str[1]);
118                       content += element.render();
119                     } else {
120                       ST element = new ST("else if(<param1>=<param2>){");
121                       element.add("param1", str[0]);
122                       element.add("param2", str[1]);
123                       content = element.render();
124                       stringList.add(content);
125                     }
126
127                   } else {
128                     String[] str = line.split("&amp;");
129                     System.out.println(str[1]);
130                     String temp = str[0].substring(8);
131                     ST element = new ST("else if(<param1>&&<param2>){");
132                     element.add("param1", temp);
133                     element.add("param2", str[1]);
134                     content = element.render();
```

```
135                     stringList.add(content);
136                   }
137
138                 } else {
139                   line = line.trim();
140                   String[] str = line.split("=");
141
142                 }
143
144               }
145
146             } else if (line.trim().startsWith("entry")) {
147               String[] str = line.split(" ");
148               UnifaceParser parser = getParserTree(str[1]);
149               ParseTree tree = parser.entry();
150               ST element = new ST("public void <name>(){");
151               element.add("name", tree.getText());
152               stringList.add(element.render());
153
154             } else if (line.trim().startsWith("end")) {
155               UnifaceParser parser = getParserTree("}");
156               ParseTree tree = parser.end();
157               ST h = new ST("<endparam>");
158               h.add("endparam", tree.getText());
159               stringList.add(h.render());
160
161             } else if (line.trim().compareTo("endif") == 0) {
162               UnifaceParser parser = getParserTree("}");
163               ParseTree tree = parser.endif();
164               ST h = new ST("<endifparam>");
165               h.add("endifparam", tree.getText());
166               stringList.add(tree.getText());
167             } else if (line.trim().startsWith("clear")) {
168               line = line.trim();
169               String[] str = line.split("\\s+");
170               String[] str1 = str[0].split("/");
171               System.out.println(str[0].substring(str[0].length() - 1));
```

```
172             UnifaceParser parser =
                ↪  getParserTree(str[0].substring(str[0].length() - 1).trim() +
                ↪  "," + str[1]);
173             ParseTree tree = parser.clear();
174             String[] str2 = tree.getText().split(",");
175             ST h = new ST("clear(<param>,<value>);");
176             h.add("param", str2[0]);
177             h.add("value", str2[1]);
178             stringList.add(h.render());
179           } else if (line.trim().startsWith("retrieve")) {
180             line = line.trim();
181             String[] str = line.split("\\s+");
182             String[] str1 = str[0].split("/");
183             UnifaceParser parser =
                ↪  getParserTree(str[0].substring(str[0].length() - 1) + "," +
                ↪  str[1]);
184             ParseTree tree = parser.retrieve();
185             String[] str2 = tree.getText().split(",");
186             ST h = new ST("retrieve(<param>,<value>);");
187             h.add("param", str2[0]);
188             h.add("value", str2[1]);
189             stringList.add(h.render());
190           } else if (line.trim().compareTo("return") == 0) {
191             UnifaceParser parser = getParserTree("return;");
192             ParseTree tree = parser.returnl();
193             ST h = new ST("<returnval>");
194             h.add("returnval", tree.getText());
195             stringList.add(h.render());
196           } else if (line.trim().startsWith("creocc")) {
197             line = line.trim();
198             String[] str = line.split("\\s+");
199             UnifaceParser parser = getParserTree(str[1]);
200             ParseTree tree = parser.creocc();
201             ST h = new ST("<creoccval>");
202             h.add("creoccval", tree.getText());
203             stringList.add(h.render());
204           } else if (line.trim().startsWith("repeat")) {
205             UnifaceParser parser = getParserTree("do{");
206             ParseTree tree = parser.dol();
```

```
207            ST h = new ST("<repeat>");
208            h.add("repeat", tree.getText());
209            stringList.add(h.render());
210
211          } else if (line.trim().startsWith("sort")) {
212            line = line.trim();
213            String[] str = line.split("\\s+");
214            String[] str1 = str[2].split("\\.");
215            String temp = str1[1] + "." + str1[0];
216            UnifaceParser parser = getParserTree("sort(e," + str[1] + temp +
               ↪  ");");
217            if (parser != null) {
218              ParseTree tree = parser.sort();
219              stringList.add(tree.getText());
220            }
221
222          } else if (line.trim().startsWith("message")) {
223            if (line.trim().indexOf("%%") > 0) {
224              line = line.trim();
225              String[] str = line.split("\\s+");
226              UnifaceParser parser = getParserTree(str[2]);
227              ParseTree tree = parser.message();
228
229              ST h = new ST("message(<messageparam>);");
230              h.add("messageparam", tree.getText());
231              stringList.add(h.render());
232            } else {
233              line = line.trim();
234              String[] str = line.split("\\s+");
235              UnifaceParser parser = getParserTree(str[2]);
236              ParseTree tree = parser.message();
237              ST h = new ST("message(<messageparam>);");
238              h.add("messageparam", tree.getText());
239              stringList.add(h.render());
240            }
241
242          } else if (line.trim().startsWith("askmess")) {
243            line = line.trim();
244            String[] str = line.split("\\s+");
```

```
245            UnifaceParser parser = getParserTree(str[1]);
246            ParseTree tree = parser.askmess();
247            ST h = new ST("askmess(<param>);");
248            h.add("param", tree.getText());
249            stringList.add(h.render());
250          } else if (line.trim().startsWith("while")) {
251            String[] str = line.trim().split("\\s+");
252            UnifaceParser parser = getParserTree(str[1]);
253            ParseTree tree = parser.while1();
254            ST h = new ST("while(<param>){");
255            h.add("param", tree.getText());
256            stringList.add(h.render());
257          } else if (line.trim().startsWith("message/error")) {
258            line = line.trim();
259            String[] str = line.split("\\s+");
260            content = "message(error," + str[1] + ");";
261            UnifaceParser parser = getParserTree(str[1]);
262            ParseTree tree = parser.messageerror();
263            ST h = new ST("messgae(error,<param>);");
264            h.add("param", tree.getText());
265            stringList.add(h.render());
266
267          } else if (line.trim().startsWith("endwhile")) {
268            UnifaceParser parser = getParserTree("}");
269            ParseTree tree = parser.endwhile();
270            ST h = new ST("<param>");
271            h.add("param", tree.getText());
272            stringList.add(h.render());
273          } else if (line.trim().indexOf("=") > 0) {
274            if (line.trim().indexOf(".") < 0) {
275              content = line + ";";
276              ST element = new ST("<content>;");
277              element.add("content", line);
278              stringList.add(element.render());
279            }
280
281          } else if (line.trim().startsWith("activate")) {
282            System.out.println(line);
283            line = line.trim();
```

```
284             String[] str = line.split("\\s+");
285             UnifaceParser parser = getParserTree(str[1]);
286             ParseTree tree = parser.endwhile();
287             ST h = new ST("activate(<param>);");
288             stringList.add(h.render());
289           } else if (line.trim().startsWith("askmess/info")) {
290             System.out.println(line);
291             line = line.trim();
292             String[] str = line.split("\\s+");
293             String[] str1 = str[0].split("/");
294             UnifaceParser parser =
              ↪   getParserTree(str[0].substring(str[0].length() - 4,
              ↪   str[0].length() - 1) + "," + str[1]);
295             ParseTree tree = parser.retrieve();
296             String[] str2 = tree.getText().split(",");
297             ST h = new ST("askmess(<param>,<value>);");
298             h.add("param", str2[0]);
299             h.add("value", str2[1]);
300             stringList.add(h.render());
301           } else if (line.trim().compareTo("else") == 0) {
302             UnifaceParser parser = getParserTree("}\nelse{");
303             ParseTree tree = parser.elsef();
304             ST element = new ST("<element>");
305             element.add("element", tree.getText());
306             stringList.add(element.render());
307           } else if (line.trim().endsWith("0")) {
308             System.out.println(line);
309             content = line + ";";
310             stringList.add(content);
311
312           } else if (line.trim().startsWith("setocc")) {
313             String[] str = line.split("\\s+");
314             String temp = "1";
315             UnifaceParser parser = getParserTree("PG_CHOICE_DET" + "," + temp);
316             ParseTree tree = parser.setocc();
317             ST h = new ST("setocc(<setoccval>,<temp>);");
318             stringList.add(h.render());
319
320           } else if (line.trim().startsWith("end;")) {
```

```
321            System.out.println(line);
322            content = "}";
323            UnifaceParser parser = getParserTree("}");
324            ParseTree tree = parser.setocc();
325            ST element = new ST("<end>");
326            element.add("end", tree.getText());
327            stringList.add(content);
328
329          } else if (line.trim().startsWith("Entry")) {
330            String[] str = line.split(" ");
331            UnifaceParser parser = getParserTree(str[1]);
332            ParseTree tree = parser.entry();
333            ST element = new ST("public void <name>(){");
334            element.add("name", tree.getText());
335            System.out.println(element.render());
336            stringList.add(element.render());
337          } else if (line.trim().startsWith("string")) {
338            content = line + ";";
339            stringList.add(content);
340          } else if (line.trim().startsWith("$2")) {
341            if (line.indexOf(";") > 0) {
342              String[] str = line.split(";");
343              content = str[0] + ";" + "//" + str[1];
344              stringList.add(content);
345            } else {
346              UnifaceParser parser = getParserTree(line + ";");
347              ParseTree tree = parser.assign();
348              stringList.add(tree.getText());
349            }
350
351          } else if (line.trim().startsWith("$10")) {
352            UnifaceParser parser = getParserTree(line.trim() + ";");
353            ParseTree tree = parser.assign();
354            stringList.add(tree.getText());
355
356          } else if (line.trim().startsWith("break")) {
357            UnifaceParser parser = getParserTree("break;");
358            ParseTree tree = parser.breakl();
359            ST element = new ST("<break>");
```

```
360                 element.add("break", tree.getText());
361                 stringList.add(element.render());
362               } else if (line.trim().startsWith("compare")) {
363                 line = line.trim();
364                 String[] str = line.split("\\s+");
365                 String[] str1 = str[0].split("/");
366                 System.out.println(str[0].substring(str[0].length() - 1));
367                 UnifaceParser parser =
                  ↪   getParserTree(str[0].substring(str[0].length() - 1).trim() +
                  ↪   "," + str[1]);
368                 ParseTree tree = parser.clear();
369                 String[] str2 = tree.getText().split(",");
370                 ST h = new ST("compare(<param>,<value>);");
371                 h.add("param", str2[0]);
372                 h.add("value", str2[1]);
373                 stringList.add(h.render());
374               }
375               if (line.trim().contains(".") == true) {
376                 if (line.indexOf("if") < 0) {
377                   content = "";
378                   String[] str = line.split("=", -1);
379                   if (str[0].indexOf(".") > 0) {
380                     String[] str1 = str[0].split("\\.", -1);
381                     try {
382                       content = str1[1].trim() + "." + str1[0].trim();
383
384                     } catch (ArrayIndexOutOfBoundsException e) {
385                       e.printStackTrace();
386                     }
387
388                   }
389                   if (str.length == 2) {
390                     if (str[1].indexOf(".") > 0) {
391                       String[] str2 = str[1].split("\\.", -1);
392                       content += "=" + str2[1].trim() + "." + str2[0].trim() + ";";
393
394                     } else {
395                       content += "=" + str[1].trim() + ";";
396
```

```
397                 }
398               }
399 
400               stringList.add(content);
401 
402             }
403           }
404         }
405       }
406 
407       bufferedReader.close();
408     } catch (FileNotFoundException ex) {
409       System.out.println(
410         "Unable to open file '" +
411         fileName + "'");
412     } catch (IOException ex) {
413       System.out.println(
414         "Error reading file '" +
415         fileName + "'");
416 
417     }
418     FileWriter writer = new FileWriter("output.txt");
419     for (String str: stringList) {
420       writer.write(str);
421       writer.write("\r\n");
422     }
423     writer.close();
424 
425   }
426 
427   private static UnifaceParser getParserTree(String str) {
428     ANTLRInputStream input = new ANTLRInputStream(str);
429 
430     UnifaceLexer lexer = new UnifaceLexer(input);
431 
432     CommonTokenStream tokens = new CommonTokenStream(lexer);
433 
434     UnifaceParser parser = new UnifaceParser(tokens);
435 
```

```
436       return parser;
437     }
438
439   }
```

Bibliography

[1] É. Lévénez, “History of programming languages.” O’Reilly, 2004.

[2] “Discovering uniface: A .net developer’s analysis - uniface white paper.” https://docplayer.net/55797701-White-paper-discovering-uniface-a-net-developer-s-analysis.html. (accessed: 2020-06-27).

[3] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, “Popularity, interoperability, and impact of programming languages in 100,000 open source projects,” in *2013 IEEE 37th annual computer software and applications conference*, pp. 303–312, IEEE, 2013.

[4] G. Canfora, M. Di Penta, and L. Cerulo, “Achievements and challenges in software reverse engineering,” *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, 2011.

[5] R. W. Floyd, “The syntax of programming languages-a survey,” *IEEE Transactions on Electronic Computers*, no. 4, pp. 346–353, 1964.

[6] K. Sartipi, K. Kontogiannis, and F. Mavaddat, “Architectural design recovery using data mining techniques,” in *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pp. 129–139, IEEE, 2000.

[7] J. W. Nutting, “Examination of modeling languages to allow quantitative analysis for model-based systems engineering,” tech. rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 2014.

[8] Uniface - Compuware Corporation, “Uniface component-based development methodology.” https://umanitoba.ca/computing/ist/internal/admin_sys/project_review/media/umet.pdf, dec 2000.

[9] M. Z. Yafi and A. Fatima, “Syntax Recovery for Uniface as a Domain Specific Language,” in *2018 UKSim-AMSS 20th International Conference on Computer Modelling and Simulation (UKSim)*, (Cambridge, United Kingdom), pp. 61–66, IEEE, 2018.

[10] “Xml schema part 2: Datatypes, second edition.” https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#typesystem. (accessed: 2020-08-29).

[11] W3s Design, *The GoF Design Patterns Reference.* W3s Design, 2017.

[12] A. A. H. Alzahrani, M. Yafi, and F. Alarfaj, “Some Considerations on UML Class Diagram Formalisation Approaches,” *World Academy of Science, Engineering and Technology, International Science Index 89, International Journal of Computer and Information Engineering*, vol. 8, no. 5, pp. 741–744, 2014.

[13] Informix Software, *Informix guide to SQL.* Prentice Hall PTR, 2000.

[14] M. G. van den Brand, M. A. Sellink, C. Verhoef, *et al.*, *Reengineering COBOL software implies specification of the underlying dialects.* Citeseer, 1997.

[15] K. Ketler and R. D. Smith, “Differences between third and fourth generation programmers: A human factor analysis,” vol. 5, no. 2, pp. 25–35. (accessed: 2015-02-07).

[16] T. Cipresso and M. Stamp, *Software Reverse Engineering*, pp. 659–696. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[17] K. Bennett, “Legacy systems: coping with success,” *IEEE Software*, vol. 12, pp. 19–23, Jan 1995.

[18] E. J. Chikofsky, J. H. Cross, *et al.*, “Reverse engineering and design recovery: A taxonomy,” *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[19] J. Harrison and A. Berglas, “Data flow analysis within the ITOC information system design recovery tool,” in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pp. 227–236.

[20] J. Sadiq and T. Waheed, “Reverse engineering & design recovery: An evaluation of design recovery techniques,” in *Computer Networks and Information Technology (ICCNIT), 2011 International Conference on*, pp. 325–332, IEEE, 2011.

[21] S. Diehl, *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media, 2007.

[22] T. Ball and S. G. Eick, “Software visualization in the large,” *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[23] B. A. Price, R. M. Baecker, and I. S. Small, “A principled taxonomy of software visualization,” *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211–266, 1993.

[24] R. Lämmel and C. Verhoef, “Semi-automatic grammar recovery,” *Software: Practice and Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.

[25] W. H. Roetzheim, “Estimating software costs,” *SOFTWARE DEVELOPMENT-SAN FRANCISCO-*, vol. 8, no. 10, pp. 66–68, 2000.

[26] G. M. Weinberg, *The psychology of computer programming*, vol. 29. Van Nostrand Reinhold New York, 1971.

[27] C. Jones, *Assessment and control of software risks.* Yourdon Press, 1994.

[28] P. Abel, *IBM PC Assembly language and programming.* Prentice Hall, 1998.

[29] IEEE, "Ieee standard for software maintenance," *IEEE Std 1219-1993*, pp. 1–45, June 1993.

[30] P. Mohagheghi, B. Anda, and R. Conradi, "Effort estimation of use cases for incremental large-scale software development," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 303–311, 2005.

[31] K. H. Davis and P. H. Alken, "Data reverse engineering: a historical survey," in *Proceedings Seventh Working Conference on Reverse Engineering*, pp. 70–78, 2000.

[32] H. A. Müller, K. Wong, and S. R. Tilley, "Understanding software systems using reverse engineering technology," in *Object-Oriented Technology for Database and Software Systems*, pp. 240–252, World Scientific, 1995.

[33] P. Kadam, S. Khamitkar, and S. Thorat, "Understanding contemporary advances in reverse engineering tools," 2014.

[34] S. Sippu and E. Soisalon-Soininen, "On ll (k) parsing," *Information and Control*, vol. 53, no. 3, pp. 141–164, 1982.

[35] N. Wirth, "The programming language pascal (revised report)," *Berichte der Fachgruppe Computerwissenschaften*, vol. 5, 1973.

[36] M. Fowler, *Domain-specific languages.* Pearson Education, 2010.

[37] E. C. dos Santos and P. Vilain, "Automated acceptance tests as software requirements: An experiment to compare the applicability of fit tables and gherkin language," in *International Conference on Agile Software Development*, pp. 104–119, Springer, 2018.

[38] B. Muschko, *Gradle in action.* Manning, 2014.

[39] N. Rawlings, "The history of nomad: A fourth generation language," *IEEE Annals of the History of Computing*, vol. 36, no. 1, pp. 30–38, 2014.

[40] Rosettacode.org, "Category:mapper - rosetta code." https://rosettacode.org/wiki/Category:MAPPER. (accessed: 2020-06-13).

[41] Uniface USA LLC, "Uniface.com." http://Uniface.com. (accessed: 2020-02-03).

[42] R. Sandgathe, "Report & query creation using cerner's dvdev, ccl and layout builder: Discern visual developer release 2008.01-volume 2," 2010.

[43] B. B. Miller, "Fourth-generation languages (4gls) and personal computers," in *Proceedings of the July 9-12, 1984, National Computer Conference and Exposition*, AFIPS '84, (New York, NY, USA), p. 555–559, Association for Computing Machinery, 1984.

[44] J. Martin, "Fourth - generation languages. volume 2. representative 4gls,"

[45] C. Kipp, *Programming Informix SQL/4GL: a step-by-step approach.* Prentice Hall PTR, 1995.

[46] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus, "Xbase: implementing domain-specific languages for java," *ACM SIGPLAN Notices*, vol. 48, no. 3, pp. 112–121, 2012.

[47] S. S. Sriparasa, *JavaScript and JSON essentials.* Packt Publishing Ltd, 2013.

[48] H. Albin-Amiot and Y.-G. Guéhéneuc, "Design patterns: A round-trip," in *15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary*, 2001.

[49] L. Lavagno, G. Martin, and B. Selic, *UML for Real.* Springer, 2003.

[50] L. Wu, H. Sahraoui, and P. Valtchev, "Coping with legacy system migration complexity," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pp. 600–609, IEEE, 2005.

[51] J. Harrison, A. Berglas, and I. Peake, "Legacy 4gl application migration via knowledge-based software engineering technology: a case study," in *Software Engineering Conference, 1997. Proceedings., Australian*, pp. 70–78.

[52] P. Kumar and K. Syed, "Software testing–goals, principles, and limitations," 2010.

[53] J. A. Lehman and J. C. Wetherbe, "A survey of 4gl users and applications," *Information System Management*, vol. 6, no. 3, pp. 44–52, 1989.

[54] A. Chandrasekran and R. Broadwater, "Closing the generation gap (4gls in the power education curriculum)," *IEEE Transactions on Power Systems*, vol. 4, no. 2, pp. 808–811, 1989.

[55] J. J. Dolado, "A study of the relationships among albrecht and mark ii function points, lines of code 4gl and effort," *Journal of Systems and Software*, vol. 37, no. 2, pp. 161–173, 1997.

[56] K. P. Arnett and M. C. Jones, "Programming languages: Today and tomorrow," *Journal of Computer Information Systems*, vol. 33, no. 4, pp. 77–81, 1993.

[57] D. D. McCracken, J. F. Nunamaker Jr, and N. Rawlings, "The role of fourth generation languages in computer science education (panel session)," in *Proceedings of the 1985 ACM*

annual conference on The range of computing: mid-80's perspective: mid-80's perspective, p. 181, 1985.

[58] S. S. Chawla, J. Chen, A. Gorelik, H. C. Thio, and D. Wang, "Specification to abap code converter," Aug. 3 2004. US Patent 6,772,409.

[59] F. R. Jacobs and D. C. Whybark, *Why Erp? A Primer on Sap Implementation.* McGraw-Hill Higher Education, 1st ed., 2000.

[60] M. Toyama, "Supersql: an extended sql for database publishing and presentation," in *ACM SIGMOD Record*, vol. 27, pp. 584–586, ACM, 1998.

[61] D. L. Monge and T. A. Schultz, "Process for providing transitive closure using fourth generation structure query language (sql)," Oct. 6 1998. US Patent 5,819,257.

[62] Uniface.com, "Uniface 10 web edition, featuring modern ide, is released | uniface." http://www.uniface.com/news-entry/uniface-10-web-edition-featuring-modern-ide-is-released/. (accessed: 2019-02-20).

[63] Q. Zhu, Y. Yang, M. Natale, E. Scholte, and A. Sangiovanni-Vincentelli, "Optimizing the software architecture for extensibility in hard real-time distributed systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 621–636, 2010.

[64] A. A. H. Alzahrani, A. H. Eden, and M. Z. Yafi, "Conformance checking of single access point pattern in JAAS using codecharts," in *2015 World Congress on Information Technology and Computer Applications (WCITCA)*, pp. 1–6, IEEE, 2015.

[65] A. A. H. Alzahrani, A. H. Eden, and M. Z. Yafi, "Structural Analysis of the Check Point Pattern," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pp. 404–408, IEEE, 2014.

[66] W. Zhao, "A language based formalism for domain driven development," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 130–131, ACM. (accessed: 2015-02-07).

[67] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[68] K. Czarnecki, "Software reuse and evolution with generative techniques," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 575–575, 2007.

[69] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.

[70] E. Evans, *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[71] A. L. Tharp, “The impact of fourth generation programming languages,” *ACM SIGCSE Bulletin 16, no. 2*, vol. 2, 1984.

[72] J. D. Allen, D. Anderson, J. Becker, R. Cook, M. Davis, P. Edberg, M. Everson, A. Freytag, L. Iancu, R. Ishida, *et al.*, “The unicode standard,” *Mountain view, CA*, 2012.

[73] M. R. Lowry, “Methodologies for knowledge-based software engineering,” in *International Symposium on Methodologies for Intelligent Systems*, pp. 219–234, Springer, 1993.

[74] L. Sharma and A. Gera, “A survey of recommendation system: Research challenges,” *International Journal of Engineering Trends and Technology (IJETT)*, vol. 4, no. 5, pp. 1989–1992, 2013.

[75] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács, “Solutions for reverse engineering 4gl applications, recovering the design of a logistical wholesale system,” in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 343–346, IEEE, 2011.

[76] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, “Columbus-reverse engineering tool and schema for c++,” in *International Conference on Software Maintenance, 2002. Proceedings.*, pp. 172–181, IEEE, 2002.

[77] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács, “Magister: Quality assurance of magic applications for software developers and end users,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–6, IEEE, 2010.

[78] S. Yamamoto, R. Kawasaki, and M. Nagaoka, “VGUIDE: 4gl application platform for large distributed information systems,” in *Computer Software and Applications Conference, 1996. COMPSAC ’96., Proceedings of 20th International*, pp. 536–541.

[79] J. V. Harrison, P. A. Bailes, A. Berglas, and I. Peake, “Re-engineering 4gl-based information system applications,” in *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*, pp. 448–457, IEEE, 1995.

[80] P. Beynon-Davies, *Computer Aided Information Systems Engineering (CAISE)*, pp. 103–108. London: Macmillan Education UK, 1998.

[81] T. J. Parr, H. G. Dietz, and W. E. Cohen, “Pccts reference manual: version 1.00,” *ACM Sigplan Notices*, vol. 27, no. 2, pp. 88–165, 1992.

[82] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[83] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[84] S. Muchnick, *Advanced compiler design implementation.* Morgan Kaufmann, 1997.

[85] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Program. Lang. Syst.*, vol. 11, p. 491–516, Oct. 1989.

[86] J. Dong, "Uml extensions for design pattern compositions," *Journal of object technology*, vol. 1, no. 5, pp. 151–163, 2002.

[87] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic code generation from design patterns," *IBM Systems Journal*, vol. 35, no. 2, pp. 151–171, 1996.

[88] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse engineering of design components," in *Proceedings of the 21st international conference on Software engineering*, pp. 226–235, ACM, 1999.

[89] R. Schauer, R. K. Keller, B. Lague, G. Knapen, S. Robitaille, and G. Saint-Denis, "The spool design repository: architecture, schema, and mechanisms," in *Advances in software engineering*, pp. 269–294, Springer, 2002.

[90] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177–1193, 2009.

[91] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Advances in Engineering Software*, vol. 41, no. 4, pp. 519–526, 2010.

[92] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pp. 153–160, 1998.

[93] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 49–54, 2007.

[94] J. Nilsson, W. Löwe, J. Hall, and J. Nivre, "Parsing formal languages using natural language parsing techniques," in *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, pp. 49–60, 2009.

[95] R. G. Cattell, J. M. Newcomer, and B. W. Leverett, "Code generation in a machine-independent compiler," *ACM SIGPLAN Notices*, vol. 14, no. 8, pp. 65–75, 1979.

[96] M. Elson and S. T. Rake, “Code-generation technique for large-language compilers,” *IBM Systems Journal*, vol. 9, no. 3, pp. 166–188, 1970.

[97] R. S. Glanville and S. L. Graham, “A new method for compiler code generation,” in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 231–254, 1978.

[98] R. G. Cattell, “A survey and critique of some models of code generation,” tech. rep., Dept. of Computer Science Carnegie-Mellon University, Pittsburg, PA, USA, 1977.

[99] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, “Code comparison system based on abstract syntax tree,” in *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pp. 668–673, IEEE, 2010.

[100] A. Veeramani, K. Venkatesan, and K. Nalinadevi, “Abstract syntax tree based unified modeling language to object oriented code conversion,” in *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*, pp. 1–8, 2014.

[101] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” in *Proceedings of the 2005 international workshop on Mining software repositories*, pp. 1–5, 2005.

[102] M. Rabinovich, M. Stern, and D. Klein, “Abstract syntax networks for code generation and semantic parsing,” *arXiv preprint arXiv:1704.07535*, 2017.

[103] P. Bourque and R. E. E. Fairley, *Guide to the software engineering body of knowledge (SWEBOK): Version 3.0.* IEEE Computer Society Press, 2014.

[104] B. Bellay and H. Gall, “An evaluation of reverse engineering tool capabilities,” *Journal of Software Maintenance*, vol. 10, no. 5, pp. 305–331, 1998.

[105] G. Canfora and M. Di Penta, “New frontiers of reverse engineering,” in *2007 Future of Software Engineering*, pp. 326–341, IEEE Computer Society, 2007.

[106] A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, “Reusability index: A measure for assessing software assets reusability,” in *New Opportunities for Software Reuse* (R. Capilla, B. Gallina, and C. Cetina, eds.), (Cham), pp. 43–58, Springer International Publishing, 2018.

[107] L. Bettini, V. Bono, and M. Naddeo, *A trait based re-engineering technique for Java hierarchies.* In Proceedings of the 6th international symposium on Principles and practice of programming in Java, 2008.

[108] R. W. Schwanke, “An intelligent tool for re-engineering software modularity,” in *Software Engineering, 1991. Proceedings., 13th International Conference on*, pp. 83–92, IEEE, 1991.

[109] I. Sommerville, *Software Engineering.* Addison-Wesley, 10th ed., 2015.

[110] H. M. Kienle, *Building reverse engineering tools with software components.* PhD thesis, University of Victoria, 2006.

[111] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering (FOSE'07)*, pp. 104–119, IEEE, 2007.

[112] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 47–60, ACM, 2000.

[113] M. Eltantawi and P. Maresca, "Logic programming and database schema in reverse engineering: Analysis and documentation for existing code in a multilanguage environment," *Engineering Applications of Artificial Intelligence*, vol. 9, no. 5, pp. 561–574, 1996.

[114] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke, *Re-engineering C++ component models via automatic program transformation.* In Reverse Engineering, 12th Working Conference on, 2005.

[115] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Operations research*, vol. 14, no. 4, pp. 699–719, 1966.

[116] B. Kullbach and A. Winter, "Querying as an enabling technology in software re-engineering," in *Proceedings of the Third European Conference on Software Maintenance and Re-engineering*, pp. 42–50, IEEE, 1999.

[117] J. Ebert, V. Riediger, and A. Winter, "Graph technology in reverse engineering–the tgraph approach," in *Proc. 10th Workshop Software Reengineering. GI Lecture NFotes in Informatics*, 2008.

[118] M. Van Den Brand, P. Klint, and C. Verhoef, "Re-engineering needs generic programming language technology.," *SIGPLAN notices*, vol. 32, no. 2, pp. 54–61, 1997.

[119] F. Perin, *Reverse engineering heterogeneous applications.* PhD thesis, University of Bern, 2012.

[120] S. Ducasse, M. Lanza, and S. Tichelaar, "The moose re-engineering environment," *Smalltalk Chronicles*, vol. 3, no. 2, 2001.

[121] D. Franke, C. Elsemann, S. Kowalewski, and C. Weise, "Reverse engineering of mobile application lifecycles," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pp. 283–292, IEEE, 2011.

[122] E. J. Byrne, "Software reverse engineering: a case study," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1349–1364, 1991.

[123] H. A. Müller, S. R. Tilley, and K. Wong, "Understanding software systems using reverse engineering technology perspectives from the rigi project," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pp. 217–226, IBM Press, 1993.

[124] P. P. Chen, "A preliminary framework for entity-relationship models.," in *ER '81: Proceedings of the Second International Conference on the Entity-Relationship Approach to Information Modelling and Analysis*, pp. 19–28, 1981.

[125] P. Castro, S. Melnik, and A. Adya, "Ado. net entity framework: raising the level of abstraction in data programming," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 1070–1072, ACM, 2007.

[126] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar, "Anatomy of the ado.net entity framework," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, (New York, NY, USA), pp. 877–888, ACM, 2007.

[127] E. J. O'Neil, "Object/relational mapping 2008: hibernate and the entity data model (edm)," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1351–1356, ACM, 2008.

[128] A. Warth and I. Piumarta, "Ometa: An object-oriented language for pattern matching," in *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, (New York, NY, USA), pp. 11–19, ACM, 2007.

[129] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, "From regular expressions to parsing expression grammars," in *Brazilian Symposium on Programming Languages*, 2011.

[130] T. Parr and K. Fisher, "Ll(*): The foundation of the antlr parser generator," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 425–436, ACM, 2011.

[131] A. Koprowski and H. Binsztok, "Trx: A formally verified parser interpreter," in *European Symposium on Programming*, pp. 345–365, Springer, 2010.

[132] B. Ford, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl," *ACM SIGPLAN Notices*, vol. 37, no. 9, pp. 36–47, 2002.

[133] L. Tratt, "Direct left-recursive parsing expression grammars," *School of Engineering and Information Sciences, Middlesex University*, 2010.

[134] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pp. 13–22, IEEE, 2001.

[135] A. M. Saeidi, J. Hage, R. Khadka, and S. Jansen, "A generic framework for model-driven analysis of heterogeneous legacy software systems," 2017.

[136] ISO, "Information technology — Object Management Group Unified Modeling Language (OMG UML) — Part 2: Superstructure," standard, International Organization for Standardization, Geneva, CH, April 2004.

[137] G. Booch, "The booch method: Notation, part i.," *COMP. LANG.*, vol. 9, no. 9, pp. 47–70, 1992.

[138] F. Losavio, A. Matteo, and F. Schlienger, "Object-oriented methodologies of coad and yourdon and booch: comparison of graphical notations," *Information and Software Technology*, vol. 36, no. 8, pp. 503–514, 1994.

[139] S. Umeda, "A reference model for manufacturing enterprise system by using object modelling technology (omt) method," *ACM Siggroup Bulletin*, vol. 18, no. 1, pp. 54–57, 1997.

[140] M. E. Fayad, W.-T. Tsai, and M. L. Fulghum, "Transition to object-oriented software development," *Communications of the ACM*, vol. 39, no. 2, pp. 108–121, 1996.

[141] G. Reggio and R. Wieringa, "Thirty one problems in the semantics of uml 1.3 dynamics," in *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)–Workshop" Rigorous Modelling and Analysis of the UML: Challenges and Limitations*, Citeseer, 1999.

[142] C. Sibertin-Blanc, N. Hameurlain, and O. Tahir, "Ambiguity and structural properties of basic sequence diagrams," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 275–284, 2008.

[143] U. S. Shah and D. C. Jinwala, "Resolving ambiguity in natural language specification to generate uml diagrams for requirements specification," *International Journal of Software Engineering, Technology and Applications*, vol. 1, no. 2-4, pp. 308–334, 2015.

[144] Object Management Group, "Unified modeling language (omg uml), v2.5.1." https://www.omg.org/spec/UML/2.5.1. (accessed: 2020-02-03), 2017.

[145] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using uml 2.0: promises and pitfalls," *Computer*, vol. 39, no. 2, pp. 59–66, 2006.

[146] S. Ceri and G. Gottlob, "Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries," *IEEE Transactions on software engineering*, no. 4, pp. 324–345, 1985.

[147] R. Cyganiak, "A relational algebra for sparql," *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, vol. 35, p. 9, 2005.

[148] M. Cadoli and T. Mancini, "Combining relational algebra, sql, constraint modelling, and local search," *Theory and Practice of Logic Programming*, vol. 7, no. 1-2, pp. 37–65, 2007.

[149] W. Kim, "On optimizing an sql-like nested query," *ACM Transactions on Database Systems (TODS)*, vol. 7, no. 3, pp. 443–469, 1982.

[150] B. Cao and A. Badia, "Sql query optimization through nested relational algebra," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 3, pp. 18–es, 2007.

[151] J. Galindo, J. M. Medina, O. Pons, and J. C. Cubero, "A server for fuzzy sql queries," in *International Conference on Flexible Query Answering Systems*, pp. 164–174, Springer, 1998.

[152] J. Cabot, R. Clarisó, and D. Riera, "Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 547–548, ACM, 2007.

[153] F. Gingras and L. V. Lakshmanan, "nd-sql: A multi-dimensional language for interoperability and olap," in *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pp. 134–145, 1998.

[154] D. Chatziantoniou and K. A. Ross, "Querying multiple features of groups in relational databases," in *VLDB*, vol. 96, pp. 295–306, 1996.

[155] A. Nayak, A. Poriya, and D. Poojary, "Type of nosql databases and its comparison with relational databases," *International Journal of Applied Information Systems*, vol. 5, no. 4, pp. 16–19, 2013.

[156] Z. Zia, A. Rashid, and K. uz Zaman, "Software cost estimation for component-based fourth-generation-language software applications," *IET software*, vol. 5, no. 1, pp. 103–110, 2011.

[157] J. M. Peeples, "A software development cost estimation model for higher level language environments," *Cyber Kebumen*, 2006.

[158] J. J. Dolado, "A validation of the component-based method for software size estimation," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 1006–1021, 2000.

[159] M. Brassard, "Component-based source code generator," May 25 2004. US Patent 6,742,175.

[160] R. Gfeller, "Upgrading of component-based application," *HSR-Hochschule für Technik Rapperswill*, p. 4.

[161] C. I. Arad, *Programming Model and Protocols for Reconfigurable Distributed Systems.* PhD thesis, KTH Royal Institute of Technology, 2013.

[162] J. Ittel, M. Cherdron, and B. Goerke, "Software component architecture," Aug. 5 2008. US Patent 7,409,692.

[163] W. T. Councill and G. T. Heineman, “Component-based software engineering: putting the pieces together,” *NY: Addison Wesley*, pp. 5–99, 2001.

[164] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[165] V. L. Narasimhan, P. Parthasarathy, and M. Das, “Evaluation of a suite of metrics for component based software engineering (cbse).,” *Issues in Informing Science & Information Technology*, vol. 6, 2009.

[166] C. Hibbs, S. Jewett, and M. Sullivan, *The art of lean software development: a practical and incremental approach*. O’Reilly Media, Inc., 2009.

[167] H. Barki, S. Rivard, and J. Talbot, “Toward an assessment of software development risk,” *Journal of management information systems*, vol. 10, no. 2, pp. 203–225, 1993.

[168] Y. I. Liou and M. Chen, “Using group support systems and joint application development for requirements specification,” *Journal of Management Information Systems*, vol. 10, no. 3, pp. 25–41, 1993.

[169] E. W. Duggan and C. S. Thachenkary, “Integrating nominal group technique and joint application development for improved systems requirements determination,” *Information & Management*, vol. 41, no. 4, pp. 399–411, 2004.

[170] M. E. Khan and F. Khan, “A comparative study of white box, black box and grey box testing techniques,” *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.

[171] A. Gunda and N. Devta-Prasanna, “Method and system for improving quality of a circuit through non-functional test pattern identification,” Dec. 2 2008. US Patent 7,461,315.

[172] M. Fourment and M. R. Gillings, “A comparison of common programming languages used in bioinformatics,” *BMC bioinformatics*, vol. 9, no. 1, p. 82, 2008.

[173] B. A. Malloy, J. F. Power, and J. T. Waldron, “Applying software engineering techniques to parser design: the development of a c# parser,” 2002.

[174] A. Derezińska and A. Szustek, “Object-oriented testing capabilities and performance evaluation of the c# mutation system,” in *IFIP Central and East European Conference on Software Engineering Techniques*, pp. 229–242, Springer, 2009.

[175] P. Singh, S. Singh, and J. Kaur, “Tool for generating code metrics for c# source code using abstract syntax tree technique,” *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 1–6, 2013.

[176] D. R. Hanson and T. A. Proebsting, “A research c# compiler,” *Software: Practice and Experience*, vol. 34, no. 13, pp. 1211–1224, 2004.

[177] Y. Lilis and A. Savidis, “A survey of metaprogramming languages,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[178] D. Pawade, A. Sakhapara, S. Parab, D. Raikar, R. Bhojane, and H. Mamania, “Literature survey on automatic code generation techniques,” *i-Manager’s Journal on Computer Science*, vol. 6, no. 2, p. 34, 2018.

[179] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[180] D. J. Salomon and G. V. Cormack, “Scannerless nslr (1) parsing of programming languages,” in *ACM SIGPLAN Notices*, vol. 24, pp. 170–178, ACM, 1989.

[181] N. Chomsky, “Formal properties of grammars,” *Handbook of Math. Psychology*, vol. 2, pp. 328–418, 1963.

[182] C. R. Cook and P. S.-P. Wang, “A chomsky hierarchy of isotonic array grammars and languages,” *Computer Graphics and Image Processing*, vol. 8, no. 1, pp. 144–152, 1978.

[183] S. Yu, *Regular Languages*, pp. 41–110. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.

[184] E. Charniak, “Statistical parsing with a context-free grammar and word statistics,” *AAAI/I-AAI*, vol. 2005, no. 598-603, p. 18, 1997.

[185] A. V. Aho and J. D. Ullman, “Translations on a context free grammar,” *Information and Control*, vol. 19, no. 5, pp. 439–475, 1971.

[186] A. Cremers and S. Ginsburg, “Context-free grammar forms,” *Journal of Computer and System Sciences*, vol. 11, no. 1, pp. 86–117, 1975.

[187] J. Gough, “The gppg parser generator,” *Queensland University of Technology-Australia*, 2008.

[188] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[189] T. Parr, S. Harwell, and K. Fisher, “Adaptive ll (*) parsing: the power of dynamic analysis,” in *ACM SIGPLAN Notices*, vol. 49, pp. 579–598, ACM, 2014.

[190] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, “Modisco: a generic and extensible framework for model driven reverse engineering,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 173–174, ACM, 2010.

[191] C. Wulf, S. Frey, and W. Hasselbring, “A three-phase approach to efficiently transform c# into kdm,” 2012.

[192] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309, ACM, 2010.

[193] T. Parr, "The reuse of grammars with embedded semantic actions," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 5–10, IEEE, 2008.

[194] "Github - antlr/grammars-v4: Grammars written for antlr v4; expectation that the grammars are free of actions.." https://github.com/antlr/grammars-v4. (accessed: 2018-12-16).

[195] H. Mössenböck, "The compiler generator coco/r user manual," 2005.

[196] D. Cook, "Gold parser builder documentation. 2009," 2009.

[197] P. Cederberg, "Grammatica: Parser generator," 2015.

[198] G. R. Economopoulos, *Generalised LR parsing algorithms.* PhD thesis, Citeseer, 2006.

[199] "Grammars for hime." https://bitbucket.org/cenotelie/hime-grams/src, 2018. (accessed: 2018-12-16).

[200] J. Bovet and T. Parr, "Antlrworks: an antlr grammar development environment," *Software: Practice and Experience*, vol. 38, no. 12, pp. 1305–1332, 2008.

[201] N. Laurent and K. Mens, "Parsing expression grammars made practical," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 167–172, 2015.

[202] R. Knöll and M. Mezini, "Pegasus: first steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 542–559, 2006.

[203] "Github - sprache: Tiny c# parser construction library." https://github.com/sprache/Sprache. (accessed: 2018-12-16).

[204] "Github - datalust/superpower: A c# parser construction toolkit with high-quality error reporting." https://github.com/datalust/superpower. (accessed: 2018-12-16).

[205] "Github - takahisa/parseq: monadic parser combinator library for c#." https://github.com/takahisa/parseq. (accessed: 2018-12-16).

[206] "Github - plioi/parsley." https://github.com/plioi/parsley. (accessed: 2018-12-16).

[207] "Github - benjamin-hodgson/pidgin: A lightweight, fast and flexible parsing library for c#, developed at stack overflow.." https://github.com/benjamin-hodgson/Pidgin. (accessed: 2018-12-16).

[208] S. Czaska, "Annoflex - an annotation-based code generator for lexical scanners." https://annoflex.github.io/index.html. (accessed: 2018-12-16).

[209] L. Thomas, "Apg… abnf parser generator." http://www.coasttocoastresearch.com, 2017.

[210] R. Corbett, "Byacc parser generator version 1.9, 1993, available via anonymous ftp at ftp. cs. berkeley. edu:/ucb/4bsd/byacc. tar."

[211] H. Yuan, "Cookcc documentation — cookcc 0.4.3 documentation." http://coconut2015.github.io/cookcc/. (accessed: 2018-12-16).

[212] S. E. Hudson, F. Flannery, C. S. Anaian, D. Wang, and A. Appel, "Cup parser generator for java. 1999," *World-Wide Web page URL: http://www. cs. princeton. edu/appel/modern/java/CUP*, 2006.

[213] M. P. Jones, "jacc: just another compiler compiler for java a reference manual and user guide," 2004.

[214] T. E. Copeland, *Generating parsers with JavaCC.* Centennial Books, 2007.

[215] G. Klein, S. Rowe, and R. Décamps, "Jflex-the fast scanner generator for java," *URL: http://www. jflex. de*, 2001.

[216] L. Quesada, F. Berzal, and J.-C. Cubero, "A model-based multilingual natural language parser—implementing chomsky's x-bar theory in modelcc," in *International Conference on Flexible Query Answering Systems*, pp. 293–304, Springer, 2013.

[217] J. Kurs, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, and J. Laval, "Petitparser: Building modular parsers," 2013.

[218] E. Gagnon, B. Menking, M. Nowostawski, K. K. Agbakpem, and K. Gergely, "Sablecc," *An Object-Oriented Compiler Framework, Master of Science, School of Computer Science, McGill University, Montreal*, 2002.

[219] E. M. Gagnon and L. J. Hendren, "Sablecc, an object-oriented compiler framework," in *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pp. 140–154, IEEE, 1998.

[220] J. Harrison and A. Berglas, "Data flow analysis with the itoc information system design recovery tool," in *Proc. of Automated Software Engineering Conference*, 1997.

[221] M. Broløs, C.-J. Johnsen, and K. Skovhede, "Occam to go translator," in *2021 IEEE Concurrent Processes Architectures and Embedded Systems Virtual Conference (COPA)*, pp. 1–8, IEEE, 2021.

[222] M. Ellims, J. Bridges, and D. C. Ince, "Unit testing in practice," in *15th International Symposium on Software Reliability Engineering*, pp. 3–13, IEEE, 2004.

[223] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[224] A. Jorgensen, S. Wort, R. LoForte, and B. Knight, *Professional microsoft sql server 2012 administration*. John Wiley & Sons, 2012.

[225] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM transactions on database systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[226] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. USA: Addison-Wesley Publishing Company, 6th ed., 2010.

[227] J. Hidders, J. Paredaens, and J. Van den Bussche, "J-logic: Logical foundations for json querying," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pp. 137–149, 2017.

[228] J. Paterson, "Iso/iec 8859-1: 1997 (e) 7-bit and 8-bit codes and their extension," *International Organization for Standardization, Joint Technical Committee no 1, Subcommittee no*, vol. 2, 1997.

[229] C.-P. M. Hsing and A. T.-i. Yaung, "Table-level unicode handling in a database engine," Dec. 7 2004. US Patent 6,829,620.

[230] T. Bray, D. Hollander, A. Layman, and J. Clark, "Namespaces in xml," *World Wide Web Consortium*, 2009.

[231] S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney, "W3c xml schema definition language (xsd) 1.1 part 1: Structures," *W3C candidate recommendation*, vol. 30, no. 7.2, p. 16, 2009.

[232] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, "Xml path language (xpath)," *World Wide Web Consortium (W3C)*, 2003.

[233] J. Clark and S. DeRose, "Xml path language (xpath) version 1.0," *World Wide Web Consortium (W3C) Recommendation 16 November 1999*, 1999.

[234] P. Suri and D. Sharma, "An id based algorithm for storing xml documents in relational datbases," *International Journal of Computer Science Issues (IJCSI)*, vol. 13, no. 1, p. 64, 2016.

[235] R. Rankins, P. Bertucci, C. Gallelli, and A. T. Silverstein, *Microsoft SQL server 2008 R2 unleashed*. Pearson Education, 2010.

[236] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik, "Enhancements to sql server column stores," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 1159–1168, ACM, 2013.

[237] W. Fan, M. Garofalakis, M. Xiong, and X. Jia, "Composable xml integration grammars," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, (New York, NY, USA), pp. 2–11, ACM, 2004.

[238] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of xml schema languages using formal language theory," *ACM Trans. Internet Technol.*, vol. 5, pp. 660–704, Nov. 2005.

[239] Handoko and J. R. Getta, "An xml algebra for online processing of xml documents," in *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, (New York, NY, USA), pp. 503:503–503:507, ACM, 2013.

[240] F. Henglein and U. T. Rasmussen, "Peg parsing in less space using progressive tabling and dynamic analysis," in *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2017, (New York, NY, USA), pp. 35–46, ACM, 2017.

[241] E. B. Duffy and B. A. Malloy, "An automated approach to grammar recovery for a dialect of the c++ language," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pp. 11–20, IEEE, 2007.

[242] A. Mooij, M. Joy, G. Eggen, P. Janson, and A. Rădulescu, "Industrial software rejuvenation using open-source parsers," in *International Conference on Theory and Practice of Model Transformations*, pp. 157–172, Springer, 2016.

[243] R. Gabriëls, D. Gerrits, and P. Kooijmans, "John W. Backus," *Early years*, vol. 1924, p. 2, 1950.

[244] M. Kubicek, O. Nierstrasz, and J. Kurš, "F# parsing expression grammar," 2016.

[245] J. Pokorný, "Functional querying in graph databases," *Vietnam Journal of Computer Science*, vol. 5, pp. 95–105, May 2018.

[246] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword proximity search on xml graphs," in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)(ICDE)*, pp. 367–378, 03 2003.

[247] S. Ceri, S. Comai, E. Damiani, P. Fraternali, and L. Tanca, "Complex queries in xml-gl," in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, SAC '00, (New York, NY, USA), pp. 888–893, ACM, 2000.

[248] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for xml," *Computer networks*, vol. 31, no. 11-16, pp. 1155–1169, 1999.

[249] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "Xml-ql: A query language for xml," 1998.

[250] W. S. Brainerd, "The minimalization of tree automata," *Information and Control*, vol. 13, no. 5, pp. 484 – 491, 1968.

[251] N. Nishida and Y. Maeda, "Narrowing Trees for Syntactically Deterministic Conditional Term Rewriting Systems," in *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)* (H. Kirchner, ed.), vol. 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 26:1–26:20, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[252] C. Teichmann, A. Venant, and A. Koller, "Efficient translation with linear bimorphisms," in *Language and Automata Theory and Applications* (S. T. Klein, C. Martín-Vide, and D. Shapira, eds.), (Cham), pp. 308–320, Springer International Publishing, 2018.

[253] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, "Left recursion in parsing expression grammars," *Science of Computer Programming*, vol. 96, pp. 177–190, 2014.

[254] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *ACM SIGPLAN Notices*, vol. 39, pp. 111–122, ACM, 2004.

[255] P. Terry, *Compiling with C# and Java.* Pearson education, Pearson/Addison-Wesley, 2005.

[256] N. Wirth, "Extended backus-naur form (ebnf)," *ISO/IEC*, vol. 14977, p. 2996, 1996.

[257] J. Jones, "Abstract syntax tree implementation idioms," in *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, pp. 1–10, 2003.

[258] J. P. Brisson, "Methods and system for converting a text-based grammar to a compressed syntax diagram," Oct. 14 1997. US Patent 5,678,052.

[259] ISO, "Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)," standard, International Organization for Standardization, Geneva, CH, December 2016.

[260] H. C. Cunningham, "A little language for surveys: constructing an internal dsl in ruby," in *Proceedings of the 46th Annual Southeast Regional Conference on XX*, pp. 282–287, ACM, 2008.

[261] U. Zdun, "A dsl toolkit for deferring architectural decisions in dsl-based software design," *Information and Software Technology*, vol. 52, no. 7, pp. 733–748, 2010.

[262] I. Damyanov and M. Sukalinska, "Domain specific languages in practice," *International Journal of Computer Applications*, vol. 115, no. 2, 2015.

[263] D. Dubé and M. Feeley, “Efficiently building a parse tree from a regular expression,” *Acta informatica*, vol. 37, no. 2, pp. 121–144, 2000.

[264] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007.

[265] H. Hosoya and B. Pierce, “Regular expression pattern matching for xml,” in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’01, (New York, NY, USA), pp. 67–80, ACM, 2001.

[266] J. Radatz, A. Geraci, and F. Katki, “Ieee standard glossary of software engineering terminology,” *IEEE Std*, vol. 610121990, no. 121990, p. 3, 1990.

[267] T. L. Alves and J. Visser, “A case study in grammar engineering,” in *International Conference on Software Language Engineering*, pp. 285–304, Springer, 2008.

[268] M. P. Ward, “Language-oriented programming,” *Software-Concepts and Tools*, vol. 15, no. 4, pp. 147–161, 1994.

[269] A. M. Şutîi, M. van den Brand, and T. Verhoeff, “Exploration of modularity and reusability of domain-specific languages: an expression dsl in metamod,” *Computer Languages, Systems & Structures*, vol. 51, pp. 48–70, 2018.

[270] M. Fowler, “Language workbenches: The killer-app for domain specific languages,” 2005.

[271] A. Johnstone, E. Scott, and M. van den Brand, “Modular grammar specification,” *Science of Computer Programming*, vol. 87, pp. 23–43, 2014.

[272] A. De Lucia, G. A. Di Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli, “Migrating legacy systems towards object-oriented platforms,” in *1997 Proceedings International Conference on Software Maintenance*, pp. 122–129, IEEE, 1997.

[273] J. Paakki, “Attribute grammar paradigms—a high-level methodology in language implementation,” *ACM Comput. Surv.*, vol. 27, pp. 196–255, June 1995.

[274] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger, “Attribute grammar-based language extensions for java,” in *ECOOP 2007 – Object-Oriented Programming* (E. Ernst, ed.), (Berlin, Heidelberg), pp. 575–599, Springer Berlin Heidelberg, 2007.

[275] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, “Silver: An extensible attribute grammar system,” *Science of Computer Programming*, vol. 75, no. 1-2, pp. 39–54, 2010.

[276] A. C. Schwerdfeger and E. R. Van Wyk, “Verifiable composition of deterministic grammars,” *ACM Sigplan Notices*, vol. 44, no. 6, pp. 199–210, 2009.

[277] D. E. Knuth, “Semantics of context-free languages,” *Mathematical systems theory*, vol. 2, pp. 127–145, Jun 1968.

[278] T. Ekman and G. Hedin, "The jastadd extensible java compiler," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, (New York, NY, USA), pp. 1–18, ACM, 2007.

[279] G. Hedin, "An introductory tutorial on jastadd attribute grammars," in *International Summer School on Generative and Transformational Techniques in Software Engineering*, pp. 166–200, Springer, 2009.

[280] R. Grimm, "Better extensibility through modular syntax," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 38–51, ACM, 2006.

[281] H. Dohrn and D. Riehle, "Wom: An object model for wikitext (tech. rep. no. cs-2011-05)," *University of Erlangen, Dept. of Computer Science*, 2011.

[282] H. Dohrn and D. Riehle, "Design and implementation of the sweble wikitext parser: unlocking the structured data of wikipedia," in *Proceedings of the 7th International Symposium on Wikis and Open Collaboration*, pp. 72–81, ACM, 2011.

[283] K. Kuramitsu, "Fast, flexible, and declarative construction of abstract syntax trees with pegs," *Journal of Information Processing*, vol. 24, no. 1, pp. 123–131, 2016.

[284] K. Kuramitsu, "Nez: practical open grammar language," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 29–42, ACM, 2016.

[285] L. Németh, *Catamorphism-based program transformations for non-strict functional languages*. PhD thesis, University of Glasgow, 2000.

[286] J. Andersen and C. Brabrand, "Syntactic language extension via an algebra of languages and transformations," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 19–35, 2010.

[287] C. Brabrand and M. I. Schwartzbach, "The metafront system: Safe and extensible parsing and transformation," *Science of Computer Programming*, vol. 68, no. 1, pp. 2–20, 2007.

[288] C. Brabrand and M. I. Schwartzbach, *Growing languages with metamorphic syntax macros*, vol. 37. ACM, 2002.

[289] J. Andersen, C. Brabrand, and D. R. Christiansen, "Banana algebra: Compositional syntactic language extension," *Science of Computer Programming*, vol. 78, no. 10, pp. 1845–1870, 2013.

[290] D. Cook, "Documentation, gold parser and documentation, gold parser activex dll," *Electronically available at: http://www.devincook.com/goldparser/doc/engine/activex/index.htm.* (accessed: 2016-02-20).

[291] D. Cook, "Gold parser builder." http://www.goldparser.org/doc/. accessed: 2016-02-20.

[292] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism sdf—reference manual," *ACM Sigplan Notices*, vol. 24, no. 11, pp. 43–75, 1989.

[293] R. Lämmel and G. Wachsmuth, "Transformation of sdf syntax definitions in the asf+ sdf meta-environment," *Electronic Notes in Theoretical Computer Science*, vol. 44, no. 2, pp. 9–33, 2001.

[294] V. Zaytsev, "Incremental coverage of legacy software languages," in *Proceedings of the Third Edition of the Programming Experience Workshop (PX/17.2)*, 2017.

[295] T. Alves and J. Visser, "Grammar-centered development of vdm support," in *Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop, Newcastle, July 2005*, p. 11, Citeseer, 2006.

[296] M. De Jonge and J. Visser, "Grammars as contracts," in *International Symposium on Generative and Component-Based Software Engineering*, pp. 85–99, Springer, 2000.

[297] M. Van Den Brand, A. Sellink, and C. Verhoef, "Current parsing techniques in software renovation considered harmful," in *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pp. 108–117, IEEE, 1998.

[298] D. M. Anderson, "Modeling and analysis of sql queries in php systems," *East Carolina University*, 2018.

[299] J. Vinju, T. v. Storm, and P. Klint, "Rascal: A domain specific language for source code analysis and manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation(SCAM)*, vol. 00, pp. 168–177, 09 2009.

[300] J. A. Bergstra, J. Heering, and P. Klint, "The algebraic specification formalism asf," *ACM Press frontier series*, 1989.

[301] R. Lämmel, "A suite of metaprogramming techniques," in *Software Languages*, pp. 335–397, Springer, 2018.

[302] M. Bravenboer and E. Visser, "Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions," *SIGPLAN Not.*, vol. 39, pp. 365–383, Oct. 2004.

[303] S. Erdweg and F. Rieger, "A framework for extensible languages," *ACM SIGPLAN Notices*, vol. 49, no. 3, pp. 3–12, 2014.

[304] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[305] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[306] A. Gutkin and S. King, "Inductive string template-based learning of spoken,"

[307] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *arXiv preprint arXiv:1703.06353*, 2017.

[308] B. Bazelli and E. Stroulia, "Wl++: a framework to build cross-platform mobile applications and restful back-ends," *International Journal of Business Process Integration and Management*, vol. 8, no. 1, pp. 1–15, 2017.

[309] R. Lämmel, "Relationship maintenance in software language repositories," *arXiv preprint arXiv:1701.08124*, 2017.

[310] C. D. Shrestha, "The jvmcsp runtime and code generator for processj in java," 2016.

[311] L. Zhou and R. Nagi, "Design of distributed information systems for agile manufacturing virtual enterprises using corba and step standards," *Journal of manufacturing systems*, vol. 21, no. 1, p. 14, 2002.

[312] M. Hardwick, D. L. Spooner, T. Rando, and K. C. Morris, "Sharing manufacturing information in virtual enterprises," *Commun. ACM*, vol. 39, pp. 46–54, Feb. 1996.

[313] L. M. Braz, "Visual syntax diagrams for programming language statements," in *Proceedings of the 8th Annual International Conference on Systems Documentation*, SIGDOC '90, (New York, NY, USA), pp. 23–27, ACM, 1990.

[314] M. Shioda, H. Iwasaki, and S. Sato, "Libdsl: a library for developing embedded domain specific languages in d via template metaprogramming," in *ACM SIGPLAN Notices*, vol. 50, pp. 63–72, ACM, 2014.

[315] D. Hollman, J. Lifflander, J. Wilke, N. Slattengren, A. Markosyan, H. Kolla, and F. Rizzi, "Darma v. beta 0.5," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[316] R. R. Curtin, S. Bhardwaj, M. Edel, and Y. Mentekidis, "A generic and fast c++ optimization framework," *arXiv preprint arXiv:1711.06581*, 2017.

[317] K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, *et al.*, "The seqan c++ template library for efficient sequence analysis: a resource for programmers," *Journal of biotechnology*, vol. 261, pp. 157–168, 2017.

[318] D. Kourounis, L. Gergidis, M. Saunders, A. Walther, and O. Schenk, "Compile-time symbolic differentiation using c++ expression templates," *arXiv preprint arXiv:1705.01729*, 2017.

[319] J. Rumbaugh, "Object-oriented analysis and design (ooad)," in *Encyclopedia of Computer Science*, pp. 1275–1279, Chichester, UK: John Wiley and Sons Ltd.

[320] R. C. Martin and M. Martin, *Agile principles, patterns, and practices in C# (Robert C. Martin).* Prentice Hall PTR, 2006.

[321] Microsoft.com, ".NET | free, cross-platform, open source." https://dotnet.microsoft.com/. (accessed: 2019-03-03).

[322] S. Wiltamuth and A. Hejlsberg, "C# language specification," *Standard ECMA-334*, 2002.

[323] C. Godsil and G. F. Royle, *Algebraic graph theory*, vol. 207. Springer Science & Business Media, 2013.

[324] P. J. Landin, "The next 700 programming languages," *Commun. ACM*, vol. 9, pp. 157–166, Mar. 1966.

[325] O. T. Buus, F. Kiros, N. Wichmann, B. Selvarajah, Z. Ahmad, *et al.*, "A framework to build an object oriented mathematical tool with computer algebra system (cas) capability," in *Semantic Computing, 2007. ICSC 2007. International Conference on*, pp. 45–52, IEEE, 2007.

[326] D. Akbar, S. A. Haq, Z. U. Khan, and Z. Ahmed, "Simple object oriented designed computer algebra system," *Journal of Computational Methods in Sciences and Engineering*, vol. 8, no. 3, pp. 195–211, 2008.

[327] A. B. Simões, *Expressiveness improvements of OutSystems DSL query primitives.* PhD thesis, Faculdade de Ciências e Tecnologia, 2013.

[328] W. A. Woods, "Transition network grammars for natural language analysis," *Communications of the ACM*, vol. 13, no. 10, pp. 591–606, 1970.

[329] J. L. Snell, "Finite markov chains and their applications," *The American Mathematical Monthly*, vol. 66, no. 2, pp. 99–104, 1959.

[330] E. Koutsofios and S. C. North, "Drawing graphs with dot," 1996.

[331] I. Chivers and J. Sleightholme, "An introduction to algorithms and the big o notation," in *Introduction to Programming with Fortran*, pp. 359–364, Springer, 2015.

[332] M. Tomita, "An efficient augmented-context-free parsing algorithm," *Computational linguistics*, vol. 13, pp. 31–46, 1987.

[333] I. D. Baxter and M. Mehlich, "Reverse engineering is reverse forward engineering," in *Proceedings of the Fourth Working Conference on Reverse Engineering*, pp. 104–113, 1997.

[334] F. Campagne, *The MPS Language Workbench, Vol. 1.* North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 1st ed., 2014.

[335] M. Voelter and V. Pech, "Language modularity with the mps language workbench," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1449–1450, IEEE, 2012.

[336] M. Voelter, "Language and ide modularization, extension and composition with mps," *Generative and Transformational Techniques in Software Engineering*, vol. 124, 2011.

[337] A. Makarkin, "Textgen." https://confluence.jetbrains.com/x/fQE-Bw, 2017. (accessed: 2020-06-03).

[338] M. Völter and S. Lisson, "Supporting diverse notations in mps'projectional editor.," in *GEMOC@ MoDELS*, pp. 7–16, 2014.

[339] D. A. Schmidt, "Programming language semantics," *ACM Comput. Surv.*, vol. 28, p. 265–267, Mar. 1996.

[340] R. Torres, T. Ludwig, J. M. Kunkel, and M. F. Dolz, "Comparison of clang abstract syntax trees using string kernels," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 106–113, IEEE, 2018.

[341] M. Zhang, J. Zhang, and J. Su, "Exploring syntactic features for relation extraction using a convolution tree kernel," in *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pp. 288–295, 2006.

[342] Microsoft.com, "C# reference - microsoft docs." https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/. (accessed: 2020-12-06).

[343] M. Chemuturi, *Mastering software quality assurance: best practices, tools and techniques for software developers.* J. Ross Publishing, 2010.

[344] J. Visser, "Visitor combination and traversal control," in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 270–282, 2001.