SOURCE CODE TO FLOWCHART CONVERTER

by

Selim Delioğlu

B.S., Computer Engineering, Boğaziçi University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2007

ABSTRACT

SOURCE CODE TO FLOWCHART CONVERTER

There are a wide variety of tools that enable people to visually design flow-charts for algorithms to generate source code or code templates for desired programming languages. A challenge can be to analyze existing source code and generate a schema that describes the algorithm independent of the underlying programming language. This schema can be used to generate a flowchart or one can further combine this with existing code generators to implement a translator that converts source code from a programming language to another one.

ÖZET

KAYNAK KODUNU AKIŞ DİYAGRAMINA ÇEVİRME

Günümüzde insanlara algoritmalar için görsel olarak akış diyagramı düzenleme imkanı veren çok çeşitli araçlar mevcuttur. Bu araçlarla istenilen programlama dilleri için kaynak kodu veya kaynak kodu şablonları yaratılabilmektedir. Bundan daha zor bir görev de varolan kaynak kodlarını analiz edip bunu programlama dilinden bağımsız olarak ifade edebilen bir şema yaratmaktır. Bu şema bilgisi ile akış diyagramı yaratılabilir veya var olan kod yaratıcı araçlar da kullanılarak bir programlama dilinden diğerine tercüme yapabilen bir uygulama yapılabilir.

TABLE OF CONTENTS

ABSTRACT . . . iii
ÖZET . . . iv
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xiv
LIST OF ABBREVIATIONS . . . xv
1. INTRODUCTION . . . 1
2. CODE VISUALIZATION . . . 3
2.1. Motivation . . . 3
2.2. Source Code Parsing . . . 4
2.2.1. Tokenizing . . . 4
2.2.2. Parsing . . . 5
2.2.3. Tokenizing and Parsing Sample . . . 8
2.3. Related Work . . . 11
2.3.1. BUILD.NET . . . 11
2.3.2. LEX - YACC . . . 11
2.3.3. Yet Another Parser Parser (YAPP) XSLT . . . 13
2.3.4. XML Representations of Source Code to Ease Tool Development 17
2.3.5. MoHCA-Java A Tool for C++ to Java Conversion Support . . . 18
2.3.6. Visustin v3 Flow chart generator . . . 19
2.3.7. Code Visual 2 Flowchart . . . 20
2.3.8. JavaML: a markup language for Java source code . . . 21
3. SOURCE CODE PARSER . . . 24
3.1. Requirements . . . 24
3.2. Design . . . 25
3.3. Grammar Definition Markup Language (GDML) . . . 26
3.3.1. References . . . 29
3.3.2. Imports . . . 30
3.3.3. Keywords . . . 31
3.3.4. TokenRegExs . . . 31

3.3.5. TokenTypes 33
3.3.6. StateDictionaries 35
3.3.7. ValidatorFunctions 36
3.3.8. ReplaceFunctions 37
3.3.9. Rules 37
3.4. Generating Parser 41
3.4.1. TokenRegExs Class 42
3.4.2. TokenTypes Class 43
3.4.3. ParserStates Class 43
3.4.4. GrammarRules Class 44
3.4.5. Main Parser Class 44
3.5. Tokenizer Engine 50
3.6. Parser Engine 50
3.7. GUI Tools 56
4. APPLICATIONS 58
4.1. C# Code Parser 58
4.2. Fortran Code Parser 58
5. CONCLUSIONS 62
APPENDIX A: GDML DESERIALIZING CLASSES 64
A.1. XmlBaseDefinition Class 64
A.2. Grammar Class 67
A.3. GrammarSection Enumeration 69
A.4. MatchQuantity Enumeration 70
A.5. MatchType Enumeration 71
A.6. XmlIdentifiableElement Class 71
A.7. XmlIdentifiableElementCollection Class 72
A.8. XmlIdentifiableReference Class 73
A.9. XmlTokenRegEx Class 74
A.10.XmlTokenType Class 74
A.11.XmlIdentifiableFunction Class 76
A.12.MatchRule Class 77
A.13.MatchRuleItem Class 79

A.14. MatchRuleItemToken Class 81
A.15. MatchRuleItemRecursive Class 82
APPENDIX B: UTILITY CLASSES 84
B.1. ParseException Class 84
B.2. GrammarValidationResult Class 84
B.3. XmlProjectDefinition Class 85
APPENDIX C: TOKENIZER AND PARSER CLASSES 87
C.1. TokenDefinition Class 87
C.2. Token Class 88
C.3. TokenCollection Class 88
C.4. BaseCodeParser Class 89
C.5. ParserStateDictionary Class 92
C.6. ParserState Class 92
C.7. IndexKeyPair Class 95
C.8. ParserStateDifference Class 96
C.9. MatchResultSlotItem Class 97
C.10. MatchResultSlot Class 98
C.11. MatchResult Class 100
APPENDIX D: C# Parser Results 103
APPENDIX E: Fortran Parser Results 111
REFERENCES 117
REFERENCES NOT CITED 120

LIST OF FIGURES

Figure 2.1. Parse tree for the arithmetic operation: 2 * (3 + 4) + 5 4

Figure 2.2. Grammar sample 6

Figure 2.3. Sample C++ switch statement 8

Figure 2.4. Emulating C++ switch statement using if-else statements 10

Figure 2.5. Sample C# HelloWorld application 10

Figure 2.6. Parse tree of C# HelloWorld application 10

Figure 2.7. Sample BNF Grammar 12

Figure 2.8. Shift-reduce (bottom-up) parsing of the expression: x + y * z . . . 13

Figure 2.9. YAPP sample grammar 15

Figure 2.10. Calling YAPP generated parser 16

Figure 2.11. YAPP result of parsing expression '123 + 456 - 789' 16

Figure 2.12. Result of parsing expression '123 + 456 - 789' in YAPP XML form 16

Figure 2.13. MoHCA-Java - System overview [12] 18

Figure 2.14. Visustin main window and sample flow chart [14] 19

Figure 2.15. Code Visual to Flowchart main window and sample flow chart [15] 20

Figure 2.16. Example JAVA class 22

Figure 2.17. JavaML script for the example JAVA class 23

Figure 3.1. SourceCodeParser framework 26

Figure 3.2. Stages of source code parsing 27

Figure 3.3. GDML sections 28

Figure 3.4. GDML References section 29

Figure 3.5. GDML Imports section 30

Figure 3.6. GDML Keywords section 31

Figure 3.7. GDML TokenRegExs section 32

Figure 3.8. GDML TokenTypes section 34

Figure 3.9. TokenValidatorDelegate prototype 35

Figure 3.10. GDML StateDictionaries section 36

Figure 3.11. GDML ValidatorFunctions section 36

Figure 3.12. MatchValidatorDelegate prototype 36

Figure 3.13. GDML ReplaceFunctions section 38

Figure 3.14. MatchReplaceDelegate prototype 39

Figure 3.15. GDML Rules section 40

Figure 3.16. Generated TokenRegExs class 42

Figure 3.17. Generated TokenTypes class 43

Figure 3.18. Generated ParserStates class 44

Figure 3.19. Generated GrammarRules class 44

Figure 3.20. Generated TokenValidator methods 45

Figure 3.21. Generated match validator methods 46

Figure 3.22. Generated match replace methods 47

Figure 3.23. Generated constructor statement for the Initialize() call 48

Figure 3.24. Generated constructor statements for keywords 48

Figure 3.25. Generated constructor statements for token types 48

Figure 3.26. Generated constructor statements for grammar rule instances . . . 49

Figure 3.27. Generated constructor statements for a grammar rule 49

Figure 3.28. Tokenizer engine 51

Figure 3.29. Rule of type “sequence” 52

Figure 3.30. Rule of type “any” 52

Figure 3.31. Unbound recursion 54

Figure A.1. XmlBaseDefinition class 64

Figure A.2. Grammar class 67

Figure A.3. GrammarSection enumeration 70

Figure A.4. MatchQuantity enumeration 70

Figure A.5. MatchType enumeration 71

Figure A.6. XmlIdentifiableElement class 72

Figure A.7. XmlIdentifiableElementCollection class 72

Figure A.8. XmlIdentifiableReference class 73

Figure A.9. XmlTokenRegEx class 74

Figure A.10. XmlTokenType class 75

Figure A.11. XmlIdentifiableFunction class 76

Figure A.12. MatchRule class 78

Figure A.13. MatchRuleItem class 80

Figure A.14. MatchRuleItemToken class 81

Figure A.15. MatchRuleItemRecursive class 82

Figure B.1. ParseException class 84

Figure B.2. GrammarValidationResult class 84

Figure B.3. XmlProjectDefinition class 85

Figure C.1. TokenDefinition class 87

Figure C.2. Token Class 88

Figure C.3. TokenCollection class 88

Figure C.4. BaseCodeParser class 89

Figure C.5. ParserStateDictionary class 92

Figure C.6. ParserState class 92

Figure C.7. IndexKeyPair class 95

Figure C.8. ParserStateDifference class 96

Figure C.9. MatchResultSlotItem class 97

Figure C.10. MatchResultSlot class 98

Figure C.11. MatchResult class 100

Figure D.1. Model of parsed source Arithmetic.cs 108

Figure D.2. Flowchart: Arithmetic.cs, IsDone() method of Arithmetic class . . 108

Figure D.3. Flowchart: Arithmetic.cs, Add() method of Arithmetic class . . . 109

Figure D.4. Flowchart: Arithmetic.cs, Main2() method of Arithmetic class . . 110

Figure E.1. Model of parsed source INTERPOLATION.f 113

Figure E.2. Flowchart: Main program of INTERPOLATION.f 114

Figure E.3. Flowchart: Subroutine TEST_SUB of INTERPOLATION.f 115

Figure E.4. Flowchart: Subroutine AITKEN of INTERPOLATION.f 116

LIST OF TABLES

Table 2.1. Tokens of C# HelloWorld application 9

Table 4.1. C# parser features 59

Table 4.2. C# parser features (continued) 60

LIST OF ABBREVIATIONS

API	Application programming interface
ASP	Active Server Pages
ASP.NET	Active Server Pages .NET
AST	Abstract syntax tree
BMP	Windows bitmap image file
BNF	Backus-Naur Form
C#	C# programming language
C++	C++ programming language
CASE	Computer-aided software engineering
COBOL	Common Business-Oriented Language
CodeDOM	Code Document object model
DOM	Document object model
DOT	DOT Graph Description Language
EMF	Microsoft Enhanced Metafile
Fortran	Formula Translating System
FSM	Finite state machine
GDML	Grammar Definition Markup Language
GIF	Graphics Interchange Format
GUI	Graphical User Interface
IDE	Integrated development environment
J#	Microsoft JAVA-Sharp programming language
JPG	Joint Photographic Experts Group
JSP	JAVA Server Pages
MSDN	Microsoft Developer Network
PDA	Push-down automaton
Perl	Practical Extraction and Report Language
PHP	PHP: Hypertext Preprocessor
PL – SQL	Procedural Language/Structured Query Language
PNG	Portable Network Graphics

PS	Postscript
$SGML$	Standard Generalized Markup Language
$T - SQL$	Transact-SQL
$VB.NET$	Visual Basic .NET programming language
VB	Visual Basic Language
VBA	Visual Basic for Applications
VBS	Visual Basic Scripting Language
$VBScript$	Visual Basic Scripting Language
WMF	Windows Metafile
XML	Extensible Markup Language
XSD	XML Schema Definition
$XSDML$	Extensible Software Document Markup Language
$XSLT$	Extensible Stylesheet Language Transformations
$XQuery$	XML Query Language

1. INTRODUCTION

Software development is a human-intensive task. Especially maintaining large software projects is very hard if correct methodologies are not applied. Some possible tasks on a software project may be maintaining, testing, validating and developing new features.

If we need to modify some part of the software, we need to navigate to that part and apply the modifications. The navigation can be improved by organizing the software and grouping it into logically related parts. One can create a source file for each class definition instead of collecting them into a single file. Also those source files can be grouped in directories such that related classes are put into the same directory. So we can browse a few directories and files to find the necessary source code. This will improve the efficiency of the involved persons but also requires them to be educated for the conventions to be applied for the source code organization.

Another possibility can be to use software that analyzes the hierarchy of the existing source code and allows navigating through that hierarchy. The hierarchy may include modules, libraries, namespaces, types, methods, members, statements etc. File based source code organization helps only to navigate to certain parts of the hierarchy. If each class definition is put in a separate file we can navigate to the class definitions easily. But we need other tools to navigate inside the class itself. We may need a tree-like navigation to browse the fields, methods, properties and other structures in the source code. Inside a method we may want to have a flowchart representation of the statements. If we have a flowchart representation we can visually follow the algorithm flow easily.

In this thesis, a generic framework for source code to flowchart conversion, SourceCodeParser is proposed. This framework defines a generic format for specifying languages. The language specification can be used to generate and compile a parser for that language. The parser may be used to convert a source code matched by the

language specification of the parser into a flowchart representation. Furthermore the generic parser engine can be used to do any user defined tasks as long as these tasks can be expressed in the form of grammar rules.

In the next chapter core concepts are introduced, code visualization is analyzed and the issues are stated. Then some related work on the topic will be presented (both scientific and commercial). In chapter 3 the design and implementation of the SourceCodeParser framework and the GUI tools of SourceCodeParser are explained in detail. Chapter 4 studies the sample applications for the SourceCodeParser framework: SourceCodeParser for the languages C# and Fortran 77, which convert source codes to a flowchart. Finally the chapter 5 contains the conclusions and future research possibilities.

2. CODE VISUALIZATION

2.1. Motivation

Software development is an intensive process of planning, implementing, testing, revising and maintaining program source codes. Reading and navigating through existing source codes is a time consuming task in large software projects. Visualizing existing source code with flowcharts and hierarchical diagrams can be used to:

- Easier navigation through the source code
- Browse the hierarchy of existing code: namespaces, classes, methods, properties etc.
- Visually manage large blocks of source code
- Create class diagrams that give an overview of existing code
- Use existing tools like BUILD.NET [1] to visually design source code with flowcharts and even convert the source code to another programming language
- Detect possible programming errors by visually investigating the generated flowchart

Code visualization can be analyzed in two parts. First the syntactical analysis creates a parse tree from the source code. In the second part the parse tree can be further processed to create an intermediate data structure like an XML [2] file. The parse tree can also be used to create diagrams or flowcharts. Another interesting possibility is to use the parsed code structure to synthesize new source code (targeting another platform or programming language).

BUILD.NET successfully implemented a general-purpose software generator for object oriented languages. Using BUILD.NET we can build and browse software models and flowcharts and generate source code from these models. The disadvantage is that we cannot use existing source codes to generate flowcharts.

This thesis proposes the SourceCodeParser framework. SourceCodeParser is a generic framework for generating parsers for any language for which we can specify a grammar. We will use that framework to implement parsers that can convert source code from a language and to a flowchart representation BUILD.NET. This can be achieved by creating the language definition for the source code's language using SourceCodeParser framework. Using that language definition the framework may run the parser engine and create the BUILD.NET **ModelDefinition** object. So we can use existing source codes for generating BUILD.NET flowcharts.

2.2. Source Code Parsing

A compiler takes as input a source code and produces object files. The first stage of compilation is code parsing. The source code is parsed using the grammar of the programming language and the output is a parse tree. As an example look at the parse tree in figure 2.1 for an arithmetic statement.

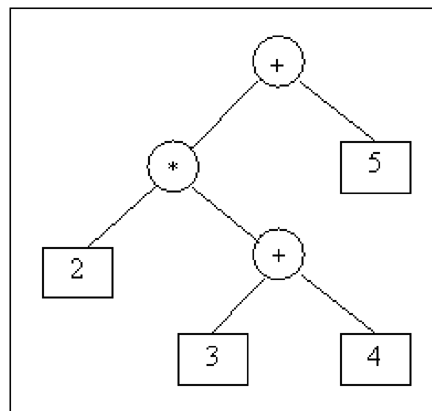


Figure 2.1. Parse tree for the arithmetic operation: 2 * (3 + 4) + 5

Parsing is the process of converting source input into a hierarchical format. The target format can be a tree of language constructs. The parsing of a source input can be implemented in two steps: tokenizing and building up the parse tree.

2.2.1. Tokenizing

In this step we determine the series of characters that build up tokens in the language. Tokens are the simplest elements of the language and are used to build up

higher level constructs. Common tokens of a language are as follows:

- **Keywords:** These are reserved words with special meaning in the language. Keywords are usually used to distinguish language constructs like classes, methods, properties, loops, branches, types and statements.
- **Identifiers:** These are words which identify or refer to user defined source elements like types, methods, properties and variables.
- **Operators:** These are tokens that define operations in the statements. The operations can be read, write, arithmetic, logic etc.
- **Punctuators:** Common uses of punctuators are defining the boundaries of language constructs or binding language elements. An example can be a "." token that can be used to access a member of an object.
- **Literals:** These are simple values or data elements of a type. Some common literals can be numbers or strings embedded inside a source code.
- **Comments:** These tokens contain descriptive information about the source code. Usually comments are ignored in the compile phase.

Usually tokenizing can be implemented with finite state machines (FSM). Regular expressions can be used instead of FSMs, since they have the same expressive power and they are easier to create and maintain. For this purpose, we need to have a set of token regular expressions for the language. The tokenizer can simply execute the token regular expressions inside the input source at the current token pointer. If one token regular expression matches, we can extract the token and move the token pointer to the end of the matched token. We can represent tokens with a string value and an associated token type label in a generic format. If we can successfully tokenize through all the input source the tokenization will be successful.

2.2.2. Parsing

The output of the tokenization process is a list of tokens. Tokens are useful only to some extent. To apply more advanced analysis, we need to detect the hierarchy of the language constructs inside the input source. The output of this step will be a tree-like

structure, that represents the parent-child relationships of the language constructs.

The parent-child relationships can be expressed as grammars. The grammar specifies the composition of language constructs in terms of other language constructs. Figure 2.2 displays a sample incomplete grammar.

The grammar is composed of grammar rules. The rules have a left side that specify the parent. The right side contains the child specifications. The children can be a selection from alternatives like the rule **Statement**, which can either be mapped to an **Assignment** or to a **BinaryOperator**. The elements on the right side can be rules and tokens. The rule **Assignment** refers to the operator token "=" and the rule **Variable** refers to the token **Identifier**. There can be quantity constraints on items. The rule **S** is mapped to any number of "Statement"s with the quantity specifier "*", which allows any number of statements to be matched by the rule.

```
S                 => Statement*
Statement         => Assignment | BinaryOperator
BinaryOperator    => SimpleExpression Operator SimpleExpression
SimpleExpression => Variable | Literal
Assignment        => Variable '=' Expression
Variable          => [Identifier]
```

Figure 2.2. Grammar sample

One of the grammar rules has special meaning. The rule **S** is the start rule of the grammar and it is supposed to match the whole input.

In the parsing step we use the grammar rules to match tokens and translate them to other constructs. Usually parsing can be implemented with push-down automata (PDA). One can either build a language specific PDA or a generic PDA which executes on grammar rules specified in a standardized format. If we can find a suitable generic format for specifying the grammars, we can create a parser that executes these generic grammar rules.

The grammar rules may have quantity constraints on items on the right side. To satisfy the rule, we may need to satisfy rule items with a specific constraint on the quantity. Common quantity constraints can be:

- **Exactly one:** Match the item exactly one time
- **Zero or one:** Match the item zero or one time. The item is optional.
- **Zero or more:** Match the item as many times as possible. We succeed even if there are no matching items.
- **One or more:** Match the item as many times as possible. We succeed if there are some matching items.

The first two quantity constraints are well defined, since the quantity to match the item is zero or one. The last two quantity constraints may be interpreted differently depending on the matching algorithm. The item is matched greedily as many times as possible on the input source. If the rule fails to match there are two alternatives. The first alternative is to simply fail the rule. Another alternative is to backtrack the rule item matches that support backtracking. So we try all alternatives to match the rule.

Now assume that we have a rule referring two other rules in sequence. The first rule item is of quantity “one or more” and may greedily match three tokens in sequence. The second rule item is of quantity “exactly one”. Assume that the second rule item can match the third token matched by the first rule item and fails to match the fourth token in sequence. If we do not allow backtracking the rule simply fails. If we allow backtracking the first rule item can be backtracked by removing the third matched token and the second rule item can match that token. So the rule succeeds by matching the three tokens.

The transformation of matched objects can target a simple tree or an application specific data type. If the target format is an application specific data type, we may also need to define the transformation operations for each rule in addition to defining the grammar rules. The transformation operation may be stateful, in which case the transformation operation may also modify the state.

At the parsing stage some issues may be encountered. The target data representation at this stage is highly application and language dependent. If we have constraints on the output format, we may be restricted to the structures of that format. It is very hard to find a generic representation that allows parsing all language's source code in a lossless way such that we support native language constructs. The issue is to convert unsupported language constructs to the supported constructs of the target format.

There are two solutions for this issue. Either those unsupported language constructs are ignored and not supported in the parsing. In this case we may abort parsing or lose information. We can also emulate those unsupported language constructs with the constructs on the target format. For example, let's assume we do not have a **switch** statement in the target format. We can emulate the **switch** statement with a series of **if - else** statements to some extent. Figure 2.4 displays a sample emulation for a C++ switch statement displayed in figure 2.3.

```
switch (condition) {
     case 1:
     case 2: cout << "This value is small!" << endl; break;
     case 3: cout << "This value is moderate!" << endl; break;
     case 4: cout << "This value is large!" << endl; break;
     default: cout << "This value is unknown!" << endl; break;
}
```

Figure 2.3. Sample C++ switch statement

2.2.3. Tokenizing and Parsing Sample

Figure 2.5 presents an example C# program. We can tokenize and parse it to have a parse tree. This source code can be tokenized according to the C# language specification [3]. The output is displayed in table 2.1.

The next stage will be to create a parse tree from the tokens. The parse tree can be seen in figure 2.6. Once we have a parse tree of a source code, we can browse the

Table 2.1. Tokens of C# HelloWorld application

Token Type	Content		Token Type	Content
Keyword	class		BlockComment	/* This is a hello world application */
Whitespace	␣		Newline	
Identifier	MainClass		Whitespace	␣␣␣␣␣
Whitespace	␣		Identifier	System
OperatorOrPunctuator	{		OperatorOrPunctuator	.
Newline			Identifier	Console
Whitespace	␣␣␣␣		OperatorOrPunctuator	.
Keyword	static		Identifier	WriteLine
Whitespace	␣		OperatorOrPunctuator	(
Keyword	void		RegularString	"Hello World"
Whitespace	␣		OperatorOrPunctuator	)
Identifier	Main		OperatorOrPunctuator	;
OperatorOrPunctuator	(		Newline	
OperatorOrPunctuator	)		Whitespace	␣␣␣␣
Whitespace	␣		OperatorOrPunctuator	}
OperatorOrPunctuator	{		Newline	
Newline			OperatorOrPunctuator	}
Whitespace	␣␣␣␣␣␣␣␣			

```
if (condition == 1 || condition == 2) {
      cout << "This value is small!" << endl;
} else if (condition == 3) {
    cout << "This value is moderate!" << endl;
} else if (condition == 4) {
    cout << "This value is large!" << endl;
} else {
    cout << "This value is unknown!" << endl;
}
```

Figure 2.4. Emulating C++ switch statement using if-else statements

```
class MainClass {
    static void Main() {
        /* This is a hello world application */
        System.Console.WriteLine("Hello World");
    }
}
```

Figure 2.5. Sample C# HelloWorld application

hierarchy of the tree and analyze it using some tools.

The parse tree is useful for browsing the code hierarchy. But we may need to visualize the code flow to have a better understanding of the underlying algorithm. The flowcharts are useful for analyzing methods in source codes. The issue we encounter here is to arrange method statements in a logical order that is human readable. For example readability can be increased if the statements do not overlap and the lines connecting the statements do not confuse the people.

```
ClassDefinition(MainClass)
+-- MethodDefinition(Main,static,void,[no arguments])
    +-- BlockComment(This is a hello world application)
    +-- MethodInvoke(type=System.Console,method=WriteLine)
        +-- MethodParameters
            +-- PrimitiveExpression(type=string, value="Hello World")
```

Figure 2.6. Parse tree of C# HelloWorld application

2.3. Related Work

Our thesis will be a complement of BUILD.NET. We will try to convert existing source code to the flow diagrams of BUILD.NET. Another possibility will be to combine those two frameworks to implement a framework that can translate source code from a language to another language.

Now we will examine some related work on graphical representation and transformation of source codes.

2.3.1. BUILD.NET

Existing application and code generators are designed for domain-specific tasks. They are used in the intermediate steps of software development process and are not intended to develop full scale applications. BUILD.NET introduces a new graphical application generation framework for object-oriented software implementation. It uses a language-neutral representation of source code through dialog forms and flow diagrams. So non-programmers can also create programs visually.

The framework can convert this graphical representation into source code in four different programming languages. The supported languages are C++, C#, J# and VB.NET.

2.3.2. LEX - YACC

Designing a compiler is very time consuming. There are tools to automate this task. Lesk [4] and Johnson [5] implemented the “lex” and “yacc” utilities to simplify compiler implementation. These tools generate C language codes for a lexical analyzer and a parser for the given language.

Lex is used to generate a lexical analyzer. The lexical analyzer uses regular expressions to match the input string and maps the matches to tokens. These expressions

can be expressed in the form of a finite state automaton. The resulting token values are used to simplify the parsing process.

Yacc takes a grammar as input. The grammar is stated in a variant of BNF (Backus Naur Form). A BNF grammar describes a context-free grammar. Figure 2.7 displays a sample BNF grammar. A grammar consist of productions that map nonterminal symbols on the left-hand side of a production to a series of terminals and nonterminals. Terminals are mapped to tokens returned by *lex*. Nonterminals describe composite language constructs.

```
r1: E -> E + E
r2: E -> E * E
r3: E -> id
```

Figure 2.7. Sample BNF Grammar

For parsing an expression we need to reduce it to a single nonterminal. This is known as bottom-up or shift-reduce parsing, and uses a stack for storing terms. Figure 2.8 displays the parsing process of the expression `x + y * z`. Terms to the left of the dot are put on the stack. The input is to the right of the dot. We start by shifting tokens onto the stack. If the top of the stack matches the right-hand side of a production, the matched tokens are popped from the stack and the left-hand side of the production is pushed on the stack. This process continues until all the input is shifted to the stack, and only the starting nonterminal remains on the stack.

There are some problems at this process. Consider the shift at step 6. Instead of shifting, we could have reduced, applying rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict and the grammar is *ambiguous*, since there is more than one possible derivation that will yield the expression. Another problem exists in the rule `E -> E + E`. This rule is ambiguous too, since we can recurse on the left or on the right.

If there are more than one rules that match the top of the stack, we may have a

```
 1 . x + y * z      shift
 2 x . + y * z      reduce(using r3)
 3 E . + y * z      shift
 4 E + . y * z      shift
 5 E + y . * z      reduce(using r3)
 6 E + E . * z      shift
 7 E + E * . z      shift
 8 E + E * z .      reduce(using r3)
 9 E + E * E .      reduce(using r2) emit multiply
10 E + E .          reduce(using r1) emit add
11 E .              accept
```

Figure 2.8. Shift-reduce (bottom-up) parsing of the expression: x + y * z

reduce-reduce conflict. We may reduce the match to more than one alternatives. *Yacc* handles these conflicts as follows. In the case of a *shift-reduce* conflict *yacc* shifts. For *reduce-reduce* conflicts, it will use the first rule in the listing. It also issues a warning message for each conflict. It is also possible to specify the operator precedences such that *yacc* can handle these correctly.

2.3.3. Yet Another Parser Parser (YAPP) XSLT

YAPP XSLT [6] is a lexical scanner and recursive descent parser generator, implemented in XSLT [7]. No language extensions or non-standard features are used apart from the nodeset() function. Grammars are expressed in XML form and transformed by the generator stylesheet into another XSLT. A lexical scanner may also be generated from the same grammar. The generated parser stylesheet has named templates for each construct in the grammar. It also incorporates any XSLT code that was declared in the grammar, and can be used to directly process input files containing unparsed text.

YAPP consists of the following parts:

- generator.xsl: the parser generator
- tokenizer.xsl: lexical scanner generator
- eliminator.xsl: left-recursion eliminator
- bnf-grammar.xml: BNF grammar in YAPP XML form

Assume that the input file is named as “grammar.xml”. “grammar.xml” is fed to “tokenizer.xsl” as an input and the output is “lexer.xsl” which is the generated lexical scanner. “grammar.xml” is also fed to “generator.xsl” as an input and the output is “parser.xsl” which is the generated recursive descent parser. Now the source input can be fed to “parser.xsl” as an input. “parser.xsl” includes the lexical analyzer “lexer.xsl”. The output is the parsed XML output corresponding to the input. Figure 2.9 shows a sample grammar for parsing additive expressions.

The grammar can be compiled into a lexical scanner and parser. Figure 2.10 shows an example usage for calling the parser from an XSL stylesheet. However, the sample parser will never succeed in parsing any additive expressions, because the rule uses left-recursion. The first part of an *AdditiveExpression* might be an *AdditiveExpression*. Hence the recursive descent parser will be stuck in infinite recursion. Left-recursive rules can be replaced with right-recursive counterparts. YAPP has the XSL stylesheet “eliminator.xsl” that can do this conversion.

Figure 2.11 shows the result of the parsing of the expression `123 + 456 - 789`. The file “bnf-grammar.xml” has generic templates that will turn term elements into a more readable form as shown in figure 2.12.

YAPP XSLT is a free software published under the GNU General Public Licence. The known issues are that it is incapable of error reporting which is a common problem with recursive descent parsers. The lexical scanner operates with first-match principle, where the first character pattern listed is matched as a token even if there is another alternative with a longer possible match.

```
<grammar>
  <terminal name="number"><any>.0123456789</any></terminal>
  <terminal name="plus"><equals>+</equals></terminal>
  <terminal name="minus"><equals>-</equals></terminal>
  <terminal name="end"></end></terminal>
  <construct name="AdditiveExpression">
    <option>
      <part name="number"/>
    </option>
    <option>
      <part name="AdditiveExpression"/>
      <part name="plus"/>
      <part name="number"/>
    </option>
    <option>
      <part name="AdditiveExpression"/>
      <part name="minus"/>
      <part name="number"/>
    </option>
  </construct>
  <construct name="Expression">
    <option>
      <part name="AdditiveExpression"/>
      <part name="end"/>
    </option>
  </construct>
</grammar>
```

Figure 2.9. YAPP sample grammar

```
<xsl:call-template name="p:Expression">
  <xsl:with-param name="in" select="'123 + 456 - 789'"/>
</xsl:call-template>
```

Figure 2.10. Calling YAPP generated parser

```
<term name="Expression">
  <term name="AdditiveExpression">
    <term name="number">123</term>
    <term name="AdditiveExpression">
      <term name="plus">+</term>
      <term name="number">456</term>
    </term>
  </term>
  <term name="end" />
</term>
```

Figure 2.11. YAPP result of parsing expression '123 + 456 - 789'

```
<Expression>
  <AdditiveExpression>
    <number>123</number>
    <AdditiveExpression>
      <plus>+</plus>
      <number>456</number>
    </AdditiveExpression>
  </AdditiveExpression>
  <end/>
</Expression>
```

Figure 2.12. Result of parsing expression '123 + 456 - 789' in YAPP XML form

2.3.4. XML Representations of Source Code to Ease Tool Development

Most conventional tools are designed to use a classical abstract syntax tree (AST). But using ASTs is not portable or extensible, since they use a proprietary API. The paper [8] proposes an XML representation of source code that can be used to achieve portability. With an XML representation of source code, one can use existing XML utilities (e.g., DOM [9], XSLT, or XQuery [10]) to examine and rewrite source code. Furthermore, the XML representation allows the maintainers and developers to add information about several relationships (e.g., method invocation or field access) resulting from semantic analysis. They can extend the prepared XML representation by defining new tags and attributes to freely annotate source code. The annotation is convenient for sharing common information and exchanging specific information.

The source code is converted to a form called XSDML [11] (Extensible Software Document Markup Language). Some characteristics of the system are as follows:

- It allows tool developers to extend the representation by defining new tags and attributes. In addition, information unnecessary for a specific tool can be easily eliminated since every code fragment is explicitly classified. This representation is therefore more acceptable to various CASE tools than the AST.
- It preserves all characters of code fragments and stores them in the textual contents of XML elements. This conversion is suitable for implementing tools displaying and manipulating concrete source code. For example, a refactoring tool or a coding checker may prefer to use the textual content while some tools may require abstract representation independent to a specific programming language. Furthermore, it is trivial to recover the original source code by removing all tags and leaving behind the textual contents of elements.
- It provides information resulting from semantic analysis in easy-to-understand and easy-to-use format. The authors are convinced that this information enables tool developers to avoid building the same or a similar analyzer from scratch. The provided information must be common and fundamental to all kinds of software tools although it is not enough to build them without supplemental information.

Conventional IDEs never make such information open from the beginning.

2.3.5. MoHCA-Java A Tool for C++ to Java Conversion Support

As Java becomes popular many people find it desirable to convert legacy C++ or C programs to Java. Sometimes this task is straight forward but sometimes there is a need for structural changes in the program.

The paper [12] suggests that a tool which performs rigorous analysis on a C++ program, providing detailed output on the changes necessary, will make conversion a much more efficient and reliable process. MoHCA-Java performs detailed analysis on a C++ abstract syntax tree (AST); the parameters of the analysis can be specified and extended very quickly and easily using a domain specific rule-based language.

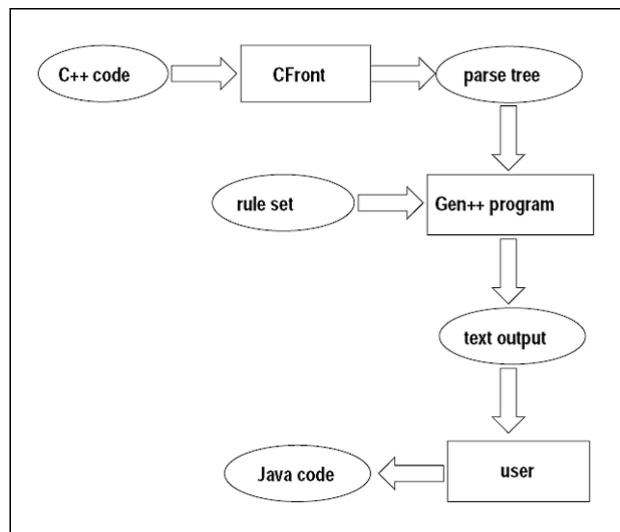


Figure 2.13. MoHCA-Java - System overview [12]

The system works as follows. The C++ code is parsed using CFront and an AST is obtained. Then Gen++ [13] runs on the parse tree with a predefined rule set. The output of this step can be used for automatic conversion. But afterwards there may be a need for human interaction for resolving issues that the system cannot handle. The capabilities of the system can be increased by developing the rule set.

2.3.6. Visustin v3 Flow chart generator

This application [14] is a commercial application for flow chart generation from existing source codes. It currently supports the following languages:

- Visual Basic, VBA, VB.NET
- C/C++, C#, Java, JavaScript
- COBOL, Fortran, Pascal/Delphi, Ada
- Perl, ASP, PHP, JSP
- T-SQL, PL/SQL

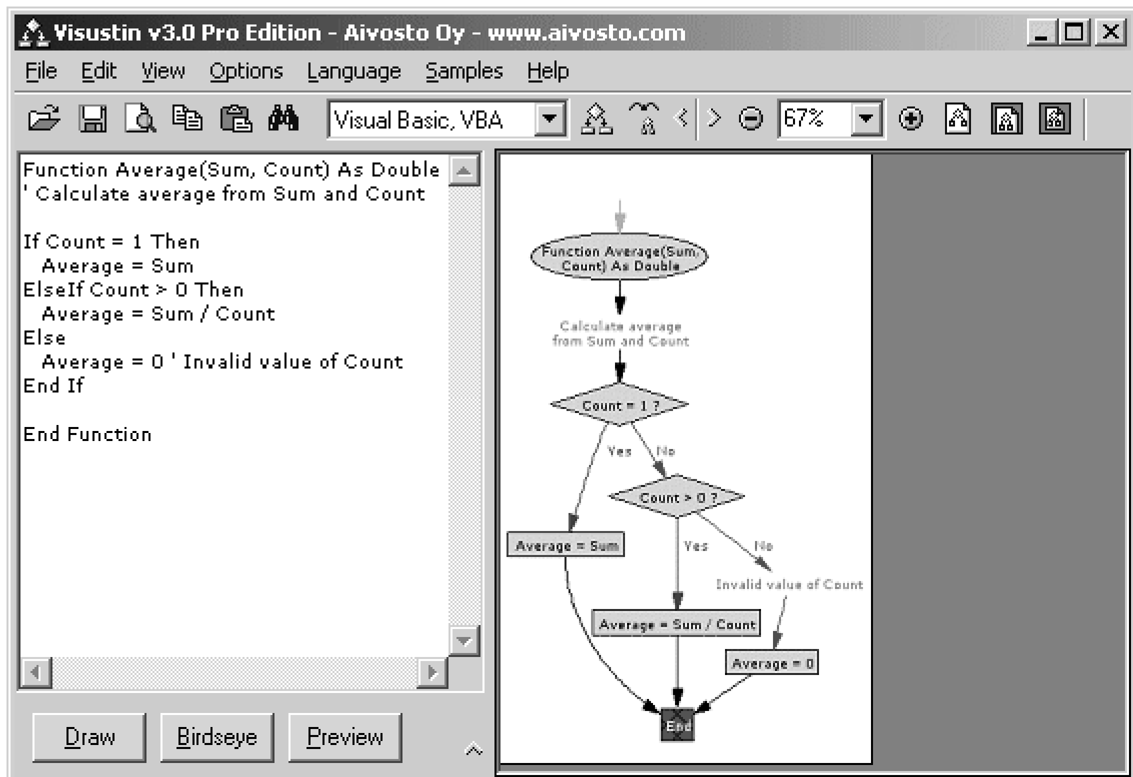


Figure 2.14. Visustin main window and sample flow chart [14]

The program has the following features:

- **Automated layout**: The program creates an optimal visual layout automatically. There is no need to adjust the charts.
- **Multi-page print**: One can preview and print large flow charts on multiple pages, or squeeze to fit on one sheet.

- **Save graphs**: The flow charts can be saved in GIF, BMP, JPG, PNG, WMF, EMF, PS or DOT image format.
- **Web publication**: Flow charts can be saved as web pages or MHT web archives.

2.3.7. Code Visual 2 Flowchart

Code Visual to Flowchart [15] is an automatic code flow chart generator software. It can analyze existing source code and generate Visio, Word, Excel, PowerPoint, PNG and BMP flowcharts. The program has a flow chart window that is synchronized with a source code editor. One can navigate through the source code and also edit the code. While browsing large flow charts, one can limit the maximum level of nested blocks.

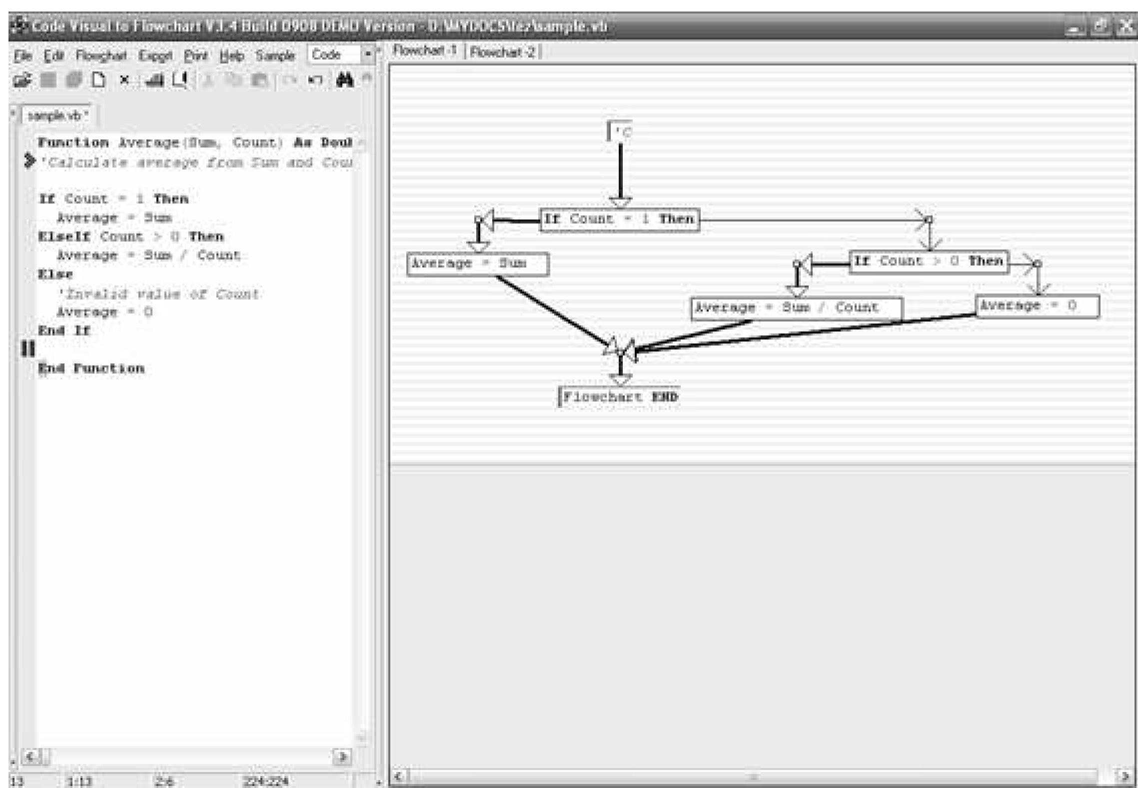


Figure 2.15. Code Visual to Flowchart main window and sample flow chart [15]

It currently supports the following languages:

- C, C++, VC++(Visual C++ .NET)
- VB(Visual Basic), VBA, QuickBasic, VBScript(VBS), ASP
- Visual C#, Visual Basic .NET, Visual J# .NET, VC++ .NET, ASP.NET

- Java, JSP
- JavaScript(JScript)
- Delphi(Object Pascal)
- PowerBuilder(PowerScript)
- PHP
- Visual Foxpro
- Peoplesoft SQR
- PL/SQL, T-SQL(Transact-sql)
- Perl

2.3.8. JavaML: a markup language for Java source code

The classical plain-text representation of source code is convenient for programmers but requires parsing to uncover the deep structure of the program. While sophisticated software tools parse source code to gain access to the program's structure, many lightweight programming aids such as grep rely instead on only the lexical structure of source code. The paper [16] describes a new XML application that provides an alternative representation of Java source code. This XML-based representation, called JavaML, is more natural for tools and permits easy specification of numerous software-engineering analysis by leveraging the abundance of XML tools and techniques. A robust converter built with the Jikes Java compiler framework translates from the classical Java source code representation to JavaML, and an XSLT stylesheet converts from JavaML back into the classical textual form.

JavaML is an alternate representation of Java source programs that is based on XML. One can parse existing JAVA code to JavaML or use XSL-transformation to convert JavaML back to JAVA source code.

Figure 2.16 shows an example JAVA class. Figure 2.17 displays the JavaML script for the sample JAVA class.

Given JavaML, the wealth of pre-existing XML and SGML [17] tools can perform

```java
import java.applet.*;
import java.awt.*;
public class FirstApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("FirstApplet", 25, 50);
    }
}
```

Figure 2.16. Example JAVA class

numerous interesting and useful analysis and transformations of Java source programs. XML tools are improving continually to support the growing infrastructure of XML based documents. Ultimately, JavaML could replace the classical source representation of Java programs as the storage format for programs, relegating text parsing to just one of many possible ways of interacting directly with the structured representation of the software artifact throughout the development process.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
<java-source-program name="FirstApplet.java">
  <import module="java.applet.*"/>
  <import module="java.awt.*"/>
  <class name="FirstApplet" visibility="public">
    <superclass class="Applet"/>
    <method name="paint" visibility="public" id="meth-15">
      <type name="void" primitive="true"/>
      <formal-arguments>
        <formal-argument name="g" id="frmarg-13">
          <type name="Graphics"/>
        </formal-argument>
      </formal-arguments>
      <block>
        <send message="drawString">
          <target>
            <var-ref name="g" idref="frmarg-13"/>
          </target>
          <arguments>
            <literal-string value="FirstApplet"/>
            <literal-number kind="integer" value="25"/>
            <literal-number kind="integer" value="50"/>
          </arguments>
        </send>
      </block>
    </method>
  </class>
</java-source-program>
```

Figure 2.17. JavaML script for the example JAVA class

3. SOURCE CODE PARSER

3.1. Requirements

BUILD.NET has implemented code generators for the following four programming languages supported by CodeDOM [18] library of Microsoft .NET Framework [19]: C++, C#, J# and VB.NET.

In our thesis we will implement a generic flexible framework that can be used to define the tokenizer and parser of a language in a generic format. The framework can generate the tokenizer and parser of the language from that definition. The generated parser is supported by the framework's execution engine and can be used to take a programming language's source code as input and return a BUILD.NET **ModelDefinition** object. We will call that framework as "SourceCodeParser". Furthermore this framework is not limited to target BUILD.NET structures. It can be used to target any user-defined data structures, since the transformations from tokens to higher level constructs are also user-defined. The generic parser engine can be used to target another set of user defined data structures and can be used in analysis tools, validation tools and conversion tools.

The generated code parser will have a function named **GetModelDefinition** that takes as input a stream of source code and returns a **ModelDefinition** object as an output. **ModelDefinition** object is the main structure of BUILD.NET for storing flowchart definitions.

The framework will be responsible for:

- Defining a general format for the language specification. We call this format Grammar Definition Markup Language (GDML).
- Defining data structures and classes to manage GDML definitions.
- Generating parser source code or the parser library from the GDML definition.

- Running a tokenizer engine and a parser engine using the GDML specification. The parser engine targets user defined data structures.
- Supporting the parsing operation using data structures that simplify the parsing operation.
- Arranging the parsed code blocks to have a standardized flowchart representation.

3.2. Design

According to the requirements, the SourceCodeParser framework consists of several parts:

- We define the GDML schema as an XML schema for creating a grammar definition.
- We define and implement the classes used to deserialize and edit GDML definitions.
- We implement the classes that generate the parser.
- We implement support classes for the tokenizer and parser engines.
- We implement some additional data structures and methods in BUILD.NET to simplify the parsing operation and arranging the flowchart.

The system layout is shown in figure 3.1.

The transformation from source code to BUILD.NET **ModelDefinition** is done by the tokenizer and parser engines of SourceCodeParser. The first part of the implementation is a tokenizer that takes input source code and returns the language specific tokens. The second part of the implementation is a parser that uses the language's grammar to parse the tokens returned from the first stage. The parse tree is used to generate the flowchart and to build the **ModelDefinition** object. Please note that those both stages are dependent on the language specification given in the GDML.

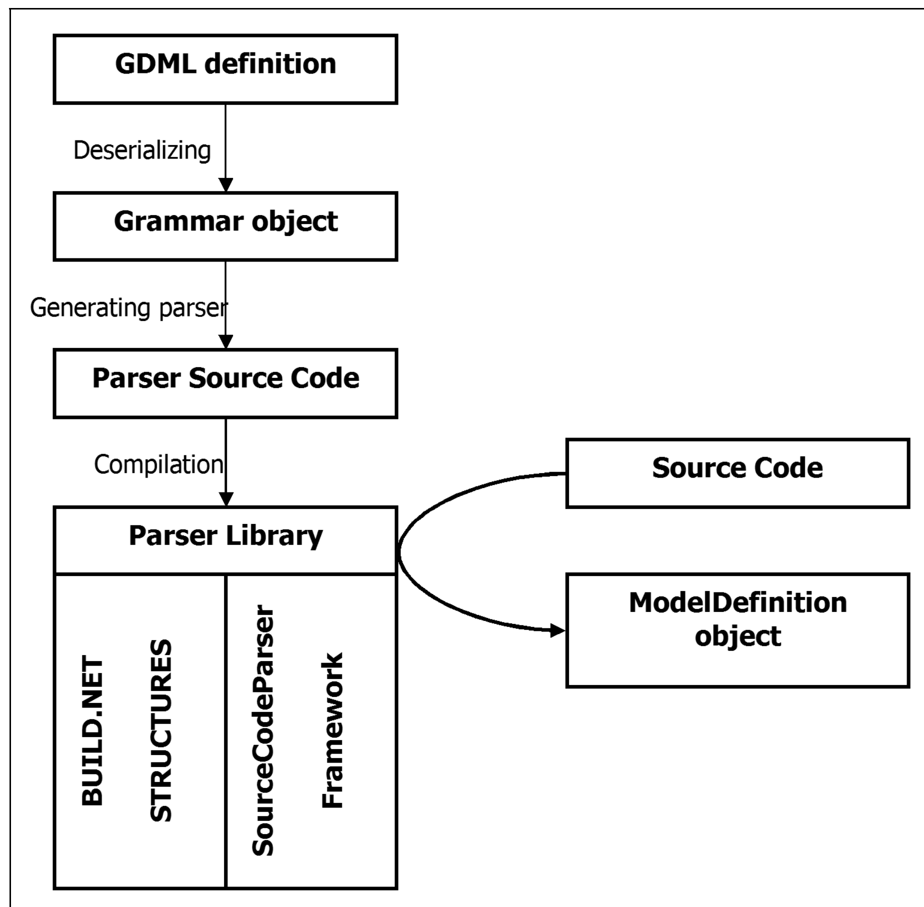


Figure 3.1. SourceCodeParser framework

3.3. Grammar Definition Markup Language (GDML)

We define Grammar Definition Markup Language (GDML) as a markup language to specify the grammar of a language. GDML is based on XML. Using GDML we basically define a tokenizer and a parser for the language. The tokenizer is defined by defining the token regular expressions and related optional token validation functions. The parser is defined by defining the grammar rules of the language. The grammar rules specify how to match groups of tokens and to transform them to higher level constructs.

SourceCodeParser takes a GDML definition as input and generates the parser for the language. The generated parser consists of several generated classes contained in a single namespace. Now we will examine GDML in detail.

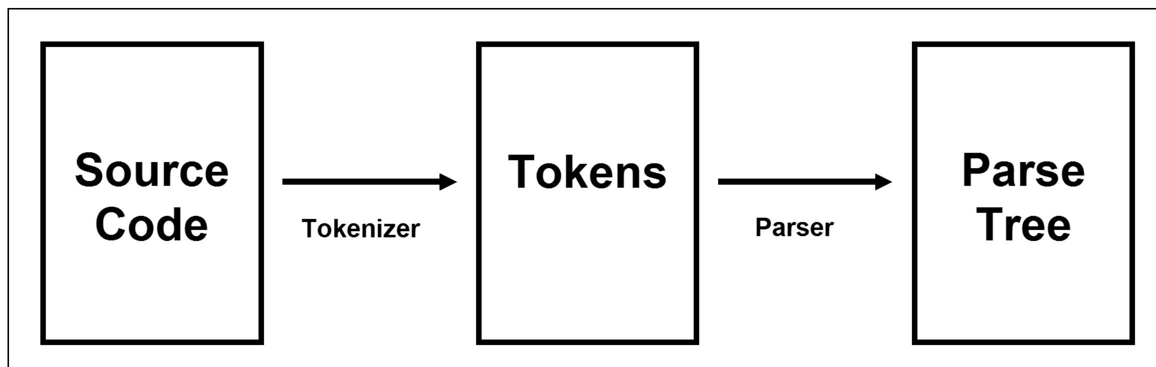


Figure 3.2. Stages of source code parsing

GDML has a root element with name **Grammar**. The root element has the following attributes:

- **targetNamespace:** Target namespace that will contain the generated parser classes.
- **targetClass:** Target class name of the generated main parser class.
- **language:** The language of the code snippets that are used in the GDML's method definitions. SourceCodeParser will use that language to generate the parser classes. The default value for this attribute is "c#".
- **startRuleID:** The identifier of the start rule of the parser. This is the entry point of the parser. The default value for this attribute is "Start".
- **caseSensitive:** This boolean flag specifies that the keywords in the language are case-sensitive. The default value for this attribute is "true".

The GDML tree consists of the following sections:

- **References:** List of assembly references for the generated parser. One can define the list of assemblies referred by the parser and the parser can use classes defined in those assemblies.
- **Imports:** List of namespace imports for the generated parser. To simplify writing GDML code snippets one can define namespace imports to use classes in those namespaces by simply using their name.
- **Keywords:** List of keywords in the language.

```
<?xml version="1.0" encoding="utf-8"?>
<Grammar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.cmpe.boun.edu.tr/grammar.xsd"
  targetNamespace="SourceCodeParser.CSharp"
  targetClass="CSharpCodeParser"
  language="c#"
  startRuleID="CompilationUnit"
  caseSensitive="true">
  <References>...</References>
  <Imports>...</Imports>
  <Keywords>...</Keywords>
  <TokenRegExs>...</TokenRegExs>
  <TokenTypes>...</TokenTypes>
  <StateDictionaries>...</StateDictionaries>
  <ValidatorFunctions>...</ValidatorFunctions>
  <ReplaceFunctions>...</ReplaceFunctions>
  <Rules>...</Rules>
</Grammar>
```

Figure 3.3. GDML sections

- **TokenRegExs:** List of regular expressions used to match tokens in the language.
- **TokenTypes:** List of token types in the language.
- **StateDictionaries:** List of names of parser state dictionaries. Those dictionaries can be used as lookup tables for language specific parse rules. For example one can define a dictionary for variable types that is accessible by the variable name.
- **ValidatorFunctions:** List of functions that will validate grammar rule matches. These optional functions behave like a filter after the rule matches.
- **ReplaceFunctions:** List of functions that will transform grammar matches to the matched objects.
- **Rules:** List of grammar rules for the language.

The sections **References** and **Imports** control the generation of the parser assembly.

The sections **Keywords**, **TokenRegExs** and **TokenTypes** define the tokenizer of the language. The tokenizer engine uses that information to tokenize the input source.

The remaining sections **StateDictionaries**, **ValidatorFunctions**, **ReplaceFunctions** and **Rules** define the parser of the language. The parser engine executes the defined parser rules and transforms the tokens gathered from the tokenizer engine to higher level language constructs.

As an example we will examine an incomplete GDML definition for the C# language.

3.3.1. References

```
<References>
  <Reference>System.dll</Reference>
  <Reference>System.Xml.dll</Reference>
  <Reference>LogicalLayer.dll</Reference>
  <Reference>SourceCodeParser.dll</Reference>
</References>
```

Figure 3.4. GDML References section

SourceCodeParser inputs a GDML file and generates a parser assembly in the .NET [20] language specified in the GDML attribute *language*. An assembly [21] is a collection of types and resources that are built to work together and form a logical unit of functionality. In the generated parser we need to access and use types and resource from other assemblies. To be able to access those types and resources, the generated parser assembly must refer to the assemblies. This section is used to define the list of those referred assemblies. A sample references section is shown in figure 3.4.

The most important assemblies to refer are:

- **System.dll**[22] which contains fundamental classes and base classes that define commonly-used types, events, interfaces, attributes, and processing exceptions.
- **System.Xml.dll**[23] which supports processing of XML data. SourceCodeParser needs this reference to be able to process XML data.
- **LogicalLayer.dll** which contains the implementation of BUILD.NET. This assembly contains the target classes of the parsing process.
- **SourceCodeParser.dll** which contains the base classes of the generated parser and implementation of SourceCodeParser itself.

3.3.2. Imports

```
<Imports>
  <Import>System</Import>
  <Import>System.Collections.Generic</Import>
  <Import>System.ComponentModel</Import>
  <Import>System.Text</Import>
  <Import>System.Xml</Import>
  <Import>System.Xml.Serialization</Import>
  <Import>BUILD.NET.LogicalLayer</Import>
  <Import>SourceCodeParser</Import>
</Imports>
```

Figure 3.5. GDML Imports section

To access types from assemblies we need to specify the full name of the type. The full name of a type is the namespace name followed by the type name. Another option to access the types is to define a set of namespace imports. So we can access the types by simply referring their name and omitting the namespace name.

In this section we can define the namespace imports that will be effective on the generated parser. We can define namespace imports to access frequently used types in

GDML code snippets easily by their name.

3.3.3. Keywords

```
<Keywords>
  <Keyword>bool</Keyword>
  <Keyword>break</Keyword>
  <Keyword>class</Keyword>
  <Keyword>const</Keyword>
  <Keyword>continue</Keyword>
</Keywords>
```

Figure 3.6. GDML Keywords section

Most of the languages have keywords and identifiers which look similar and which can be matched by similar or even the same token regular expressions. If this is the case, we need to have a list of known keywords to efficiently distinguish keywords and identifiers.

This section can be used to define keywords of the language. SourceCodeParser stores those keywords in a lookup dictionary in the generated parser. The parser code snippets can query this dictionary by using the `BaseCodeParser.IsKeyword()` function. Usually this will be done on the token validator function of the token type for language keywords. If the GDML attribute *caseSensitive* is set to “false” the keywords are matched case-insensitive.

3.3.4. TokenRegExs

This section contains the token regular expressions. Token regular expressions are identified by the element name **RegEx**. The **ID** attribute contains the unique identifier for the token regular expression. Token regular expression are composed of parts. The regular expression parts are contained in the element **Parts**. Each part can be either a simple one with text content or a reference to another token regular

```
<TokenRegExs>
  <RegEx ID="NewLine">
    <Parts>
      <Part>(\r\n?|\n)</Part>
    </Parts>
  </RegEx>
  <RegEx ID="NonNewLineChar">
    <Parts>
      <Part>[^\r\n]</Part>
    </Parts>
  </RegEx>
  <RegEx ID="EOLComment">
    <Parts>
      <Part>(//</Part>
      <Part ID="NonNewLineChar" />
      <Part>*</Part>
      <Part ID="NewLine" />
      <Part>)</Part>
    </Parts>
  </RegEx>
  <RegEx ID="Identifier">
    <Parts>
      <Part>(@?[_a-zA-Z][_a-zA-Z0-9]*)</Part>
    </Parts>
  </RegEx>
</TokenRegExs>
```

Figure 3.7. GDML TokenRegExs section

expressions. The **ID** attribute of the **Part** element defines the identifier of the referred token regular expression. The value of the token regular expression is the composite of the text contents and the values of the referred token regular expressions of each part.

If a part both defines text content and refers to a token regular expression, the text content has higher precedence.

Token regular expressions are .NET Framework Regular Expressions [24].

In the example on figure 3.7 we see that the regular expression named **NonNewLineChar** has a single part with text content "`[^\r\n]`". We also see that the regular expression named **EOLComment** is a composite of 3 simple regular expressions with text content and 2 referred regular expressions named **NonNewLineChar** and **NewLine**. The value of regular expression named **EOLComment** is "`(//[^\r\n]*)(\r\n?\|\n)`".

3.3.5. TokenTypes

This section contains the definitions of the language specific token types as shown in figure 3.8. A token is matched by a regular expression and can be validated by an optional token validator method.

A token type is defined with an **TokenType** element. The **ID** attribute contains the unique identifier of the token type. The **RegExID** attribute refers to a token regular expression from the section **TokenRegExs**. In the tokenizing phase we use that regular expression to match the token strings.

The optional **Summary** element describes the token type and the optional **Example** element can be used to specify an example of the token type.

An optional token validator method can be specified in the **Validator** element which post-validates the match. The token validator method has the prototype shown in figure 3.9.

Parameter *buffer* is the text content of the full source code. Parameter *offset* stands for the offset of match in the parameter *buffer*. *tokenString* is the potential

```
<TokenTypes>
  <TokenType ID="BoolLiteral" RegExID="BoolLiteral">
    <Summary>A boolean literal (true, false)</Summary>
  </TokenType>
  <TokenType ID="Keyword" RegExID="Identifier">
    <Summary>A reserved keyword of the language</Summary>
    <Example>class, while, for, if</Example>
    <Validator>return this.IsKeyword(tokenString);</Validator>
  </TokenType>
  <TokenType ID="Identifier" RegExID="Identifier">
    <Summary>An identifier: a variable or type</Summary>
  </TokenType>
  <TokenType ID="BlockComment" RegExID="BlockComment"
   IsSignificant="false">
    <Summary>Block comment</Summary>
    <Example>/* This is a block comment */</Example>
  </TokenType>
  <TokenType ID="WhiteSpace" RegExID="WhiteSpaces"
   IsSignificant="false">
    <Summary>A series of white space characters</Summary>
  </TokenType>
</TokenTypes>
```

Figure 3.8. GDML TokenTypes section

```
public delegate bool TokenValidatorDelegate(
    ref string buffer,
    int offset,
    ref string tokenString);
```

Figure 3.9. TokenValidatorDelegate prototype

token string that is matched at position *offset* in *buffer*. The function must validate the match and return true if the match is accepted and false otherwise.

In the sample the token types **Keyword** and **Identifier** both refer to the same token regular expression named **Identifier**. The difference is that the token type **Keyword** has a validator method which calls `BaseCodeParser.IsKeyword()` function. `BaseCodeParser.IsKeyword()` looks up the matched text from a dictionary of the known keywords of the language. So we only match the known keywords of the language as a **Keyword** token and the other words are matched as **Identifier** tokens.

3.3.6. StateDictionaries

This section contains the list of names of parser state dictionaries as shown in figure 3.10. The parser state is used in the parsing stage and may contain several dictionaries. The validator and replace functions may use these dictionaries to store and query additional state information in the parse phase.

For example, we can store the types and scopes of each variable, argument and type fields in a dictionary. If we encounter an identifier inside a method, we can find out whether that identifier is a variable, argument or a type field. We also can find the type of the variable, argument or type field. So we can create the appropriate code element for that expression in the parse phase.

```
<StateDictionaries>
  <Dictionary>VariableTypes</Dictionary>
  <Dictionary>VariableScopes</Dictionary>
</StateDictionaries>
```

Figure 3.10. GDML StateDictionaries section

```
<ValidatorFunctions>
  <Function ID="DummyValidator">
    <Code><![CDATA[
    // a dummy validator function that always succeeds
    return true;
    ]]></Code>
  </Function>
</ValidatorFunctions>
```

Figure 3.11. GDML ValidatorFunctions section

3.3.7. ValidatorFunctions

This section contains a list of functions that will validate the match for grammar rules as shown in figure 3.12. These optional functions behave like a filter after the grammar rule matches. A validator function is identified by the **Function** element. The **ID** attribute contains the unique identifier of the function. The source code of the function is stored in the **Code** element. The prototype of the functions is shown in figure 3.12.

```
public delegate bool MatchValidatorDelegate(
    ParserState state,
    MatchResult match);
```

Figure 3.12. MatchValidatorDelegate prototype

Parameter *state* is the current parser state object. The parser state object can be used to store and query user defined state information in the parsing process. Parameter *match* contains information on the block of objects matched by the rule.

The validator function must validate the match and return true if the match is accepted and false otherwise.

3.3.8. ReplaceFunctions

This section contains the list of functions that will transform grammar matches to the matched objects. A replace function is identified by the **Function** element. The **ID** attribute contains the unique identifier of the function. The source code of the function is stored in the **Code** element. The prototype of the functions is shown in figure 3.14.

Parameter *state* is the current parser state object. The parser state object can be used to store and query user defined state information in the parsing process. Parameter *match* contains information on the block of objects matched by the rule.

The function must transform the matched range of objects and return the transformed object. An example for binary operator grammar rule (i.e. a+5) is specified as the function named **BinaryOperator** in figure 3.13.

3.3.9. Rules

This section contains the grammar rules of the language. A grammar rule represents a transformation of objects in a list and is identified by a **Rule** element. The **ID** attribute contains the unique identifier of the rule. The **matchType** attribute of the **Rule** element can be either **sequence** (default) or **any**. A grammar rule has the following sections:

1. **Summary:** This optional section can be used to describe the rule.

```
<ReplaceFunctions>
  <!-- The function that will replace a binary operator match
       to a BinaryOperatorExpression -->
  <Function ID="BinaryOperator">
    <Code><![CDATA[
      BinaryOperatorExpression tmp=new BinaryOperatorExpression();
      tmp.LeftExpression = (Expression)match.Item(0);
      tmp.RightExpression = (Expression)match.Item(2);
      string operatorValue = match.TokenValue(1);
      // get the operator name according to the operator value
      tmp.OperatorName = GetOperatorName(operatorValue);
      return tmp;
    ]]></Code>
  </Function>
  <Function ID="IntegerLiteral">
    <Code><![CDATA[
      PrimitiveExpression pe = new PrimitiveExpression();
      pe.ValueType = "System.Int32";
      pe.PrimitiveValue = match.TokenValue(0);
      return pe;
    ]]></Code>
  </Function>
</ReplaceFunctions>
```

Figure 3.13. GDML ReplaceFunctions section

```
public delegate object MatchReplaceDelegate(
    ParserState state,
    MatchResult match);
```

Figure 3.14. MatchReplaceDelegate prototype

2. **Match:** This section defines how to match the objects in the input list. The section contains a sequence (sequential list) or a selection (branching) of items or rules to match.

   If **matchType** attribute contains the default value "sequence" or does not contain any value, the behavior is that the rule succeeds if the sequence of rule items in this section all succeed. If **matchType** attribute has the value "any" the rule succeeds if any of the rule items succeed at the current position in the TokenCollection.

   The **Match** section can contain the following match items:

   - **Token** which can match a token of the language by type and optional value constraint
   - **Recursive** which can refer to another rule that must succeed in the token list.

   Each match item has a **quantity** attribute. It represents the quantity to match the item, such that the match item succeeds. It can be any of the following values:

   - "1" specifies to match the item exactly one time.
   - "?" specifies to match the item zero or one time (optional item).
   - "*" specifies to match the item zero or more times. We match greedily as many times as possible.
   - "+" specifies to match the item one or more times. We match greedily as many times as possible.

   If the **quantity** attribute value is missing "1" is the default value.

3. **Validator:** This optional section defines a method to post-validate the match. If this section is specified, it can be used to filter out unwanted matches that succeed in the **Match** section.

```
<Rules>
  <Rule ID="Commas">
    <Match>
      <Token tokenType="Operator" token="," quantity="+" />
    </Match>
  </Rule>
  <Rule ID="BinaryOperatorOp" matchType="any">
    <Match>
      <Token tokenType="Operator" token="*" />
      <Token tokenType="Operator" token="+" />
      <Token tokenType="Operator" token="-" />
      <Token tokenType="Operator" token="/" />
    </Match>
  </Rule>
  <Rule ID="BinaryOperator">
    <Match>
      <Recursive ID="Expression" />
      <Recursive ID="BinaryOperatorOp" />
      <Recursive ID="Expression" />
    </Match>
    <Validator ID="DummyValidator" />
    <Replace ID="BinaryOperator" />
  </Rule>
</Rules>
```

Figure 3.15. GDML Rules section

This section can contain inline code with prototype **MatchValidatorDelegate** or refer to a predefined function defined in the section **ValidatorFunctions** using the **ID** attribute.

4. **Replace:** This section defines a method for the transformation operation of the objects that match and optionally validate successfully.

This section can contain inline code with prototype **MatchReplaceDelegate** or refer to a predefined function defined in the section **ReplaceFunctions** using the **ID** attribute.
If this section does not define inline code and does not refer to a function defined in the section **ReplaceFunctions**, the default behavior is that the matched items will be ignored in the parsing process and do not propagate to caller rules as matched objects. To achieve the same effect, one can return a "null" result as the functions return value.

In the **sequence** mode we match each rule item in sequence until all rule items match. In the sample the **BinaryOperator** rule matches 3 items in sequence:

- A block that matches the rule **Expression**
- A block that matches the rule **BinaryOperatorOp**
- A block that matches the rule **Expression**

In the **any** mode we only match the first item in the items list that succeeds at the current position. In the sample the rule **BinaryOperatorOp** matches if one of the four operator tokens "*", "+", "-" or "/" exist at the current position to match.

3.4. Generating Parser

Once we have the language defined in the form of GDML, we need to convert the specification stored in the GDML to an executable form to be able to execute the tokenizer and the parser for that language.

The **SourceCodeParser** framework contains support for a generic tokenizer and parser engine. To execute the tokenizer we need to have the token definitions. To execute the parser we need to have the grammar rules. So we must convert the GDML to an executable format that contains both token definitions and grammar rules. This is done by generating classes which contain the token regular expressions, token definitions, token validator methods, rule validator methods, rule replace methods and

grammar rules in a convenient hierarchy.

The parser is generated in two steps. In the first step the GDML definition is deserialized as a **Grammar** object. This is done using the **XmlSerializer** [25] class in the Microsoft .NET Framework. The details of classes involved in the grammar deserialization process can be found in appendix A.

Once we deserialize the GDML definition as a **Grammar** object, we can generate the parser. The **Grammar** class contains methods to generate the parser as a source code or as a compiled assembly. Now we will examine the items contained in the generated parser.

3.4.1. TokenRegExs Class

TokenRegExs is a class definition in the target namespace that contains the constants of the token regular expressions as constant string fields. A sample generated from the **TokenRegExs** section in figure 3.7 is shown in figure 3.16.

```
public class TokenRegExs {
  public const string NewLine = "(\\r\\n?|\\n)";
  public const string NonNewLineChar = "[^\\r\\n]";
  public const string EOLComment = (((("(//"
    + TokenRegExs.NonNewLineChar)
    + "*")
    + TokenRegExs.NewLine)
    + ")");
  public const string Identifier = "(@?[_a-zA-Z][_a-zA-Z0-9]*)";
}
```

Figure 3.16. Generated TokenRegExs class

3.4.2. TokenTypes Class

TokenTypes is a class definition in the target namespace that contains enumeration constants for the language token definitions. A sample generated from the **TokenTypes** section in figure 3.8 is shown in figure 3.17.

```
public class TokenTypes {
  ///<summary>A boolean literal (true, false)</summary>
  public const int BoolLiteral = 0;

  ///<summary>A reserved keyword of the language</summary>
  ///<example>class, while, for, if</example>
  public const int Keyword = 1;

  ///<summary>An identifier: a variable or type</summary>
  public const int Identifier = 2;

  ///<summary>Block comment</summary>
  ///<example>/* This is a block comment */</example>
  public const int BlockComment = 3;

  ///<summary>A series of white space characters</summary>
  public const int WhiteSpace = 4;
}
```

Figure 3.17. Generated TokenTypes class

3.4.3. ParserStates Class

ParserStates is a class definition in the target namespace that contains enumeration constants for the parser state dictionaries. A sample generated from the **StateDictionaries** section in figure 3.10 is shown in figure 3.18.

```
public class ParserStates {
  ///<summary>Parser state dictionary: VariableTypes</summary>
  public const int VariableTypes = 0;

  ///<summary>Parser state dictionary: VariableScopes</summary>
  public const int VariableScopes = 1;
}
```

Figure 3.18. Generated ParserStates class

3.4.4. GrammarRules Class

GrammarRules is a class definition in the target namespace that contains each grammar rule as a static field. A sample generated from the **StateDictionaries** section in figure 3.15 is shown in figure 3.19.

```
public class GrammarRules {
  ///<summary>Rule Commas</summary>
  public static MatchRule Commas;

  ///<summary>Rule BinaryOperatorOp</summary>
  public static MatchRule BinaryOperatorOp;

  ///<summary>Rule BinaryOperator</summary>
  public static MatchRule BinaryOperator;
}
```

Figure 3.19. Generated GrammarRules class

3.4.5. Main Parser Class

The main parser class is generated in the target namespace with the target parser class name specified in the GDML attribute **targetClass**. The parser class extends

the base class **BaseCodeParser** which contains runtime support for the tokenizing and parsing engine of **SourceCodeParser**.

The generated main parser class contains the following items:

The token validator functions are contained as methods in the parser class with a name prefix `TokenValidator_`. So if the token has the name **Keyword** the token validator method is named as `TokenValidator_Keyword`. The token validators generated from the sample in figure 3.8 are shown in figure 3.20.

```
/// <summary>
/// Token validator for tokens of type Keyword.
/// </summary>
private bool TokenValidator_Keyword(ref string buffer,
    ref int offset, ref string tokenString)
{
    return this.IsKeyword(tokenString);
}
```

Figure 3.20. Generated TokenValidator methods

The match validator functions are contained as methods in the parser class with a name prefix `MatchValidator_`. So if the validator has the name **MyValidator** the match validator method is named as `MatchValidator_MyValidator`. The match validator functions generated from the sample in figure 3.11 are shown in figure 3.21.

The match replace functions are contained as methods in the parser class with a name prefix `MatchReplace_`. The match replace functions generated from the sample in figure 3.13 are shown in figure 3.22.

The match validator functions that are defined inline in grammar rules are named with a name prefix `MatchValidatorInline_`. If the rule has the identifier `MyRule` the validator method is named as `MatchValidatorInline_MyRule`.

```
/// <summary>
/// Grammar match validator function: DummyValidator.
/// </summary>
private bool MatchValidator_DummyValidator(
    ParserState state, MatchResult match)
{
    // a dummy validator function that always succeeds
    return true;
}
```

Figure 3.21. Generated match validator methods

The match replace functions that are defined inline in grammar rules are named with a name prefix `MatchReplaceInline_`.

`List<TokenDefinition> tokenDefs`: This is a collection of token definitions. It associates selected token regular expressions with the specified token identifiers as they are specified in the GDML section **TokenTypes**. A token definition may have a token validator. In that case the token validator method is stored as a delegate in the token definition.

`List<MatchRule> rules`: This is a collection of grammar rules. It contains the information stored in the **Rules** section of the GDML. The rule contains rule items which match either tokens or other grammar rules. Those rule items are also generated and added to the rule. Also the rule's validator and replace methods are stored as delegates in the rule definition.

The generated code parser class has the following generated constructor statements:

- One statement calls the base class's **Initialize** method to set the capacities of the collections before the items are added to them. Also the boolean value of

```
/// <summary>
/// Grammar match replace function: BinaryOperator.
/// </summary>
private object MatchReplace_BinaryOperator(
    ParserState state, MatchResult match)
{
    BinaryOperatorExpression tmp = new BinaryOperatorExpression();
    tmp.LeftExpression = (Expression)match.Item(0);
    tmp.RightExpression = (Expression)match.Item(2);
    string operatorValue = match.TokenValue(1);
    // get the operator name according to the operator value
    tmp.OperatorName = GetOperatorName(operatorValue);
    return tmp;
}

/// <summary>
/// Grammar match replace function: IntegerLiteral.
/// </summary>
private object MatchReplace_IntegerLiteral(
    ParserState state, MatchResult match)
{
    PrimitiveExpression pe = new PrimitiveExpression();
    pe.ValueType = "System.Int32";
    pe.PrimitiveValue = match.TokenValue(0);
    return pe;
}
```

Figure 3.22. Generated match replace methods

the **CaseSensitive** flag is set by this method call. The statement for the sample GDML is shown in figure 3.23.

- There are statements that register the keywords. The statements for the sample

```
// Call base class's Initialize() method.
base.Initialize(true, 5, 5, 2, 3);
```

Figure 3.23. Generated constructor statement for the Initialize() call

```
// Initialize list of keywords.
base.AddKeyword("bool");
base.AddKeyword("break");
base.AddKeyword("class");
base.AddKeyword("const");
base.AddKeyword("continue");
```

Figure 3.24. Generated constructor statements for keywords

```
// Initialize list of token definitions.
base.AddTokenDefinition(TokenTypes.BoolLiteral,
  TokenRegExs.BoolLiteral);
base.AddTokenDefinition(TokenTypes.Keyword,
  TokenRegExs.Identifier, true, this.TokenValidator_Keyword);
base.AddTokenDefinition(TokenTypes.Identifier,
  TokenRegExs.Identifier);
base.AddTokenDefinition(TokenTypes.BlockComment,
  TokenRegExs.BlockComment, false);
base.AddTokenDefinition(TokenTypes.WhiteSpace,
  TokenRegExs.WhiteSpaces, false);
```

Figure 3.25. Generated constructor statements for token types

in figure 3.6 is shown in figure 3.24.

- There are the statements that register the specified token types. The statements for the sample in figure 3.8 is shown in figure 3.25.
- There are the statements that construct the grammar rule instances and assign

```
// *******************************************
// Create the grammar rule instances
// *******************************************
GrammarRules.Commas = new MatchRule("Commas");
GrammarRules.BinaryOperatorOp = new MatchRule("BinaryOperatorOp");
GrammarRules.BinaryOperator = new MatchRule("BinaryOperator");
// *******************************************
// Assign the start rule
// *******************************************
base.startRule = GrammarRules.CompilationUnit;
```

Figure 3.26. Generated constructor statements for grammar rule instances

```
// *******************************************
// Grammar rule: BinaryOperator
// *******************************************
tempRule = GrammarRules.BinaryOperator;
tempRule.Validator = this.MatchValidator_DummyValidator;
tempRule.Replace = this.MatchReplace_BinaryOperator;
tempRule.Add(new MatchRuleItemRecursive(GrammarRules.Expression));
tempRule.Add(new MatchRuleItemRecursive(
    GrammarRules.BinaryOperatorOp));
tempRule.Add(new MatchRuleItemRecursive(GrammarRules.Expression));
base.AddRule(tempRule);
```

Figure 3.27. Generated constructor statements for a grammar rule

the start rule of the grammar. The statements for the sample in figure 3.15 is shown in figure 3.26.

- There are statements that initialize the rule properties of the created rule instances. The statements for the sample rule **BinaryOperator** in figure 3.15 is shown in figure 3.27.

Once the parser class is generated, the engine can execute the tokenizer and grammar rules by creating an instance of the generated main parser class.

3.5. Tokenizer Engine

This engine takes source code as input and applies the token regular expressions at the cursor position of tokenizing process. If a token regular expression from the list of token definitions match and optionally validate the input string at the current position, the match is registered as a token to the collection of tokens. Then the cursor location is moved to the end of the match and the operation is repeated until the whole source code is tokenized.

The tokenizer engine is the method named **TokenizeSource** of the class **BaseCodeParser** (see appendix C.4 for the class BaseCodeParser). The tokenizer is shown in figure 3.28.

A token obtained by the tokenizing process is represented by a **Token** object (see appendix C.2). The tokens are collected in a **TokenCollection** object (see appendix C.3), which is simply a collection of **Token** objects.

The parser engine takes the **TokenCollection** object obtained from the tokenizer as input and executes on that collection.

3.6. Parser Engine

This engine takes a **TokenCollection** object as input. We begin applying the grammar's entry rule referred by the attribute **StartRuleID** of the grammar. The parser engine executes a recursive-descent (top-down) parser starting from the entry rule.

A grammar rule has two main parts. The first part matches and optionally validates some constraint on the input token collection. The second part applies a

```
protected TokenCollection TokenizeSource(string source) {
  TokenCollection tokens = new TokenCollection();
  int offset = 0, length = source.Length;

  while (offset < length) {
      Token tokenMatch = null;
      foreach (TokenDefinition tokenDef in this.tokenDefs)
          // the first token type that matches at the
          // current position succeeds.
          if ((tokenMatch = tokenDef.GetTokenAt(ref source,
               ref offset)) != null)
              break;

      if (tokenMatch == null)
          throw new ParseException("Cannot match a token at " +
              source.Substring(offset));
      else
          tokens.Add(tokenMatch);
  }
  return tokens;
}
```

Figure 3.28. Tokenizer engine

transformation on the matched objects.

The grammar rules are instances of the **MatchRule** class and contain one or more rule items.

The match constraint can have two types. The type is specified by the property **MatchType** of the **MatchRule** object. This property is of the enumeration type **MatchType**.

If the property has the value **Sequence** the engine tries to satisfy each rule item of the rule in a sequence to match the rule successfully. Figure 3.29 displays a sample rule match of type “sequence”.

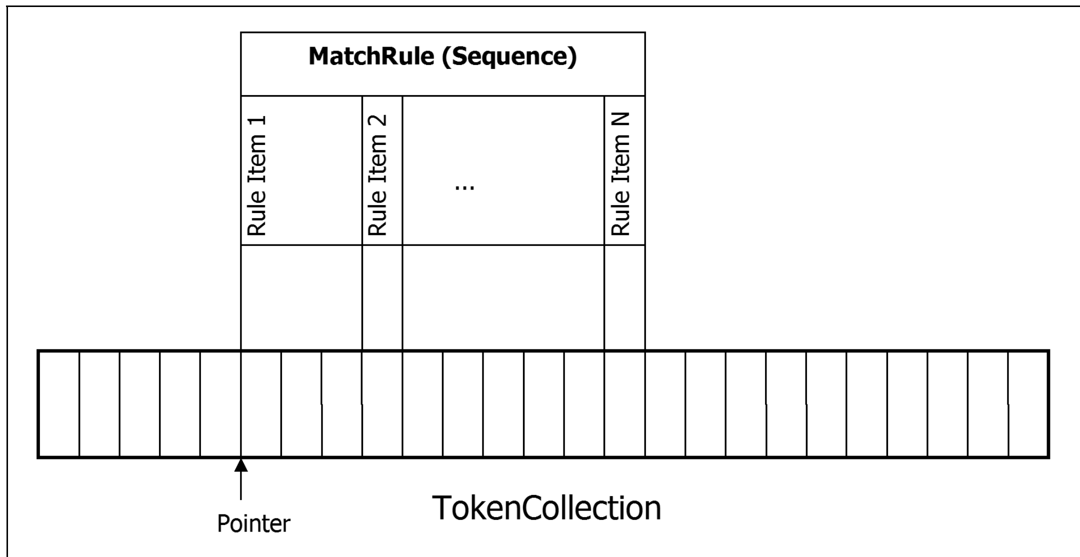


Figure 3.29. Rule of type “sequence”

If the property has the value **Any** the engine tries to satisfy any one of the rule items of the rule to match the rule successfully. In this mode the rule acts as a selection from different matching alternatives.

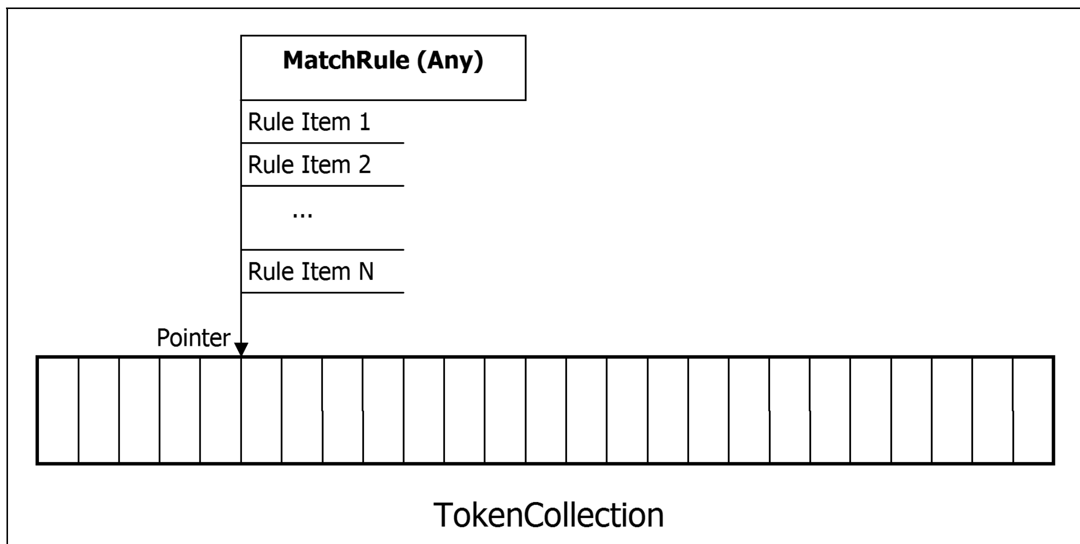


Figure 3.30. Rule of type “any”

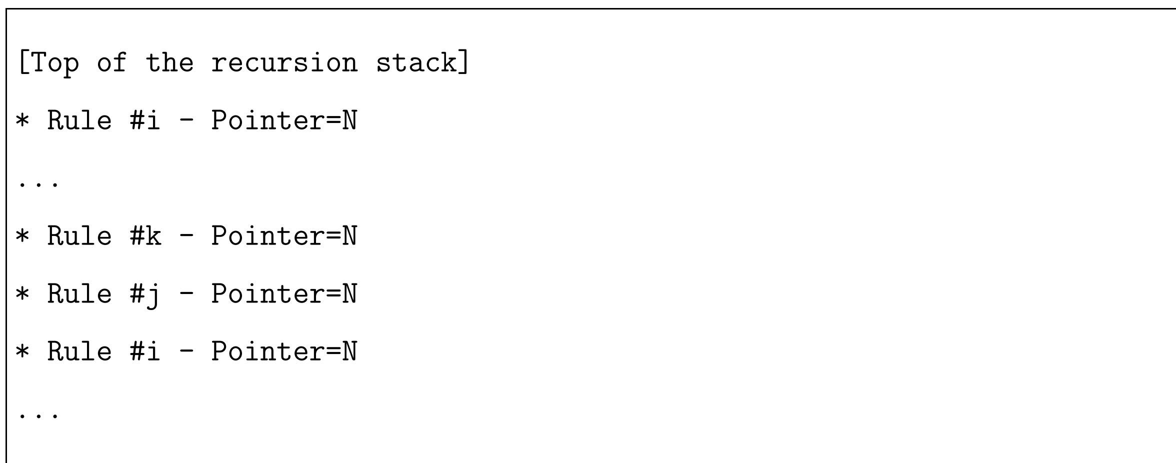
The rule items can be of type **MatchRuleItemToken** or **MatchRuleItemRecursive**.

If the rule item is of type **Token** the engine looks for tokens of specific type and of specific value (if it is specified) that matches the given quantity constraint. If the rule item is a recursive one of the type **Recursive**, the engine recursively runs the referred rule and tries to match that rule to satisfy the current rule item.

The objects matched by a match rule are stored in a **MatchResult** object. The **MatchResult** object contains **MatchResultSlot** objects for each of the rule items. The objects matched by a rule item are stored in a **MatchResultSlot** object. The number of objects matched by a rule item may be more than one, if the quantity specifier is "ZeroOrMore" or "OneOrMore". The rule item may also succeed by matching no objects, if the quantity specifier is "ZeroOrOne" or "ZeroOrMore". Each matched object is stored as a **MatchResultSlotItem** object in the **MatchResultSlot**.

There is a runtime check to avoid unbound recursions. An unbound recursion occurs if the pointer of an an executed rule is the same as the pointer of the execution of the same rule in the current recursion stack. This means that the pointer will not move if the recursion is allowed to continue. This will lead to a stack overflow. To avoid the unbound recursion, we keep track of the last pointer of execution in the token collection for each rule in the recursion stack. If we detect for a rule that there is a calling instance of the same rule executed at the same pointer in the recursion stack, we have entered an unbound recursion. Figure 3.31 displays an unbound recursion case with a recursion stack. If an unbound recursion occurs the parser engine throws an exception and ends the parsing process.

The match type "any" is designed to be a selection from matching one of the alternative constraints. Ideally those alternatives must have the quantity specifier "ExactlyOne". If a rule item of a rule of match type "any" allows zero quantity to be matched it will always match and the next rule items are ignored. So we do not allow those alternative constraints to have the quantity specifiers "ZeroOrMore" and "ZeroOrOne" to enforce higher quality definitions. The engine disallows those quantity specifiers for items of rules of type "any" and throws an exception.



```
[Top of the recursion stack]
* Rule #i - Pointer=N
...
* Rule #k - Pointer=N
* Rule #j - Pointer=N
* Rule #i - Pointer=N
...
```

Figure 3.31. Unbound recursion

If the rule item is of type **Token** the matched objects are the tokens matched by the token type and token value if the token value is specified. If we try to match a token and the token at the current pointer does not match the constraints, the engine skips any number of insignificant tokens until it finds a matching token. The token types that are specified as insignificant in the GDML can be skipped in that phase. After skipping the rule item fails, if the first significant token found does not match the constraint of the rule item.

If the rule item is of type **Recursive** the matched objects are the objects returned from the execution of the referred rule, which can also be a null reference. If the returned object is a null reference, it is ignored by the engine and removed from the **MatchResult** object before the replacement operation is executed.

In the "sequence" mode the rule matches if all of the rule items match in a sequence. In the "any" mode the rule matches if one of the rule items match and the next rule items are not executed. If the rule matches successfully and the optional validator does not reject the match, the engine runs the replacement operation (transformation) of the rule. The replacement operation is user defined, which allows total control of the transformation operation. The replacement operations are of type **MatchReplaceDelegate** and take the **MatchResult** object containing the matched objects and the active **ParserState** object as input and returns a single object that substitutes the matched objects.

The validator functions and replace functions may use the **ParserState** object to add and query state variables. The state variables added to the state in a replace operation of a recursive rule item are available in the execution of the following rule items in the rule and in their recursive rule executions. If a rule matches, the state variables added by the rule items are automatically removed from the state before the replacement operation of the rule is executed. This avoids state changes from propagating from rule items to the rules. State changes are only effective for following rule items at the same level and their recursively referred rule executions.

The **ParserState** object contains mainly two types of state information.

- **Named items:** The named items are like global variables called by their name.
- **Dictionary items:** The dictionary items are referred by a dictionary index and a name. They represent categories of state information. The available categories are defined in the GDML section **StateDictionaries**. The predefined indices of the dictionaries can be referred from the enumeration constants in the generated class **ParserStates** in the generated parser.

The parser state has a collection for user defined log items which can be used to return diagnostic information. The log items are handled differently than the state variables. If a rule matches the state variables added by the rule items are automatically removed from the state, but log items are not removed. Log items are only removed from the state if the rule fails to match.

The parser engine executes greedily with backtracking capability. The engine matches rule items as many times as possible. If a rule item fails to match for a rule of type “sequence” we try to backtrack a previous rule item by removing one matched object from its corresponding **MatchResultSlot** object. This is allowed if the quantity specifier is “ZeroOrMore” or “ZeroOrOne” and we have some matched objects in the slot or if the quantity specifier is “OneOrMore” and we have more than one matched objects in the slot. If a rule item does not allow backtracking, the engine rolls back the state changes of the entire slot and tries to backtrack the previous rule item. If a rule

item allows backtracking, the backtracking operation updates the execution pointer and rolls back the state changes applied for matching the removed object. After the backtracking the engine executes the rule item following the rule item that allowed backtracking at the updated execution pointer in the token collection.

A rule and a rule item match a range of the input token collection, which can also be empty. When executing rule items in a sequence, the pointer in the token collection is moved if a rule item matches a non-empty range. The next rule item is matched at the new pointer in the token collection. The range matched by a rule is the range of the matched rule item for rules of type “any” and the union of the ranges of the rule items for rules of type “sequence”. The parser succeeds if the entry rule of the grammar succeeds and it matches the entire range of the token collection. The parser also succeeds if the range matched by the entry rule can be extended to the entire range of the token collection by including insignificant tokens left after the matched range.

The parser engine of **SourceCodeParser** is recursive-descent and is similar to YAPP XSLT parser generator. The difference is that the runtime engine is not a generic XSLT processor and can be customized to incorporate error handling and additional necessary features. The additional features can be used to improve the efficiency of the parser and to simplify writing rules. In fact the design separates the grammar rule definition from the parser engine. So it is theoretically possible to implement other kinds of parser engines running over the same GDML definition. The left-recursion problem exists also in **SourceCodeParser**, but there is currently no control for left-recursion on parser generation time. There is a runtime check to avoid unbound recursions caused by left-recursive rules. The user must eliminate left-recursions manually to be able to parse successfully.

3.7. GUI Tools

We have GUI Tools for SourceCodeParser. These tools support editing GDML definitions, generating parser source and assembly and executing the parser on input

source files. The following functionality is included in the GUI Tools:

- Loading and saving a GDML definition file
- Editing of global GDML attributes like **TargetNamespace** and **TargetClass**
- Inserting, editing and removing of items in GDML sections using their specific editors
- Validating the GDML definition for common errors in the development.
- Loading and saving a project for running the GDML on a sample input
- Generating the parser source code and assembly
- Running the tokenizer on a sample input
- Running the tokenizer and parser on a sample input

4. APPLICATIONS

Using SourceCodeParser, we developed some applications. The aim of the sample applications are to convert an input source from different languages to an instance of **ModelDefinition** class of BUILD.NET.

4.1. C# Code Parser

This parser targets the C# language of the Microsoft .NET Platform. It is based on the C# language documentation. The parser supports the features as shown in table 4.1. There are some global limitations on some structures. These limitations are caused by the supported features of BUILD.NET and CodeDOM. One can improve the support by extending the supported features of BUILD.NET. These limitations are:

- The comments bound to elements are ignored. For example the comments before the method header are not bound to the method itself.
- Features not supported by BUILD.NET are either converted to some supported structures or cause an error in parsing. For example the statements **while**, **do-while**, **for** and **foreach** are implemented using **IterationStatement**.

A sample C# source code and its generated flowcharts can be found in appendix D.

4.2. Fortran Code Parser

The Fortran Code Parser is based on Fortran 77. Currently it can parse basic structures of a Fortran program. The supported features are:

- The program is placed in a static class named **Global** inside a namespace named **Fortran**. This is used to emulate procedural programming.
- Program block is parsed and converted to a **Main** function.

Table 4.1. C# parser features

Feature	**Support**	**Known limitations and missing features**
Namespaces	Yes	
Namespace imports	Yes	Using alias directive is not supported.
Classes	Yes	
Enumerations	Yes	Enumeration base types are not supported. Modifiers not supported except public, protected, private and internal. Initializing enumeration members is not supported.
Interfaces	Yes	Modifiers not supported except public, protected, private and internal.
Structures	Yes	
Delegates	Yes	Delegate modifiers not supported except public, protected, private and internal.
Methods	Yes	Parameter attributes are not supported.
Properties	Yes	Property get/set methods do not support attributes.
Fields	Yes	Constant fields are supported. Fields can be initialized only using primitive expressions.
Events	Yes	Event declarations with add/remove methods are not supported. Event modifiers are not supported except public, protected, private and internal.
Arrays	Yes	In an array dimension, no dimension seperators are supported.
Local constants	Yes	Local constants are converted to variables.
Attributes	Yes	Attribute targets and named attribute arguments are not supported.
if-else statement	Yes	
while statement	Yes	*while* statement is emulated using *for* statements.

Table 4.2. C# parser features (continued)

Feature	**Support**	**Known limitations and missing features**
do-while statement	Yes	Emulated using *for* statements.
for statement	Yes	*for* statement does not support more than one initializer statements and more than one iterator statements.
foreach statement	Yes	Emulated using *for* statements.
goto statement	Yes	*goto case* and *goto default* statements are not supported.
throw statement	Yes	
try-catch statement	Yes	Generic catch clauses are not supported.
Constructors	No	
Static constructors	No	
Destructors	No	
Indexer declarations	No	
Operator overloading	No	
add/remove accessors	No	
params	No	Parameter arrays to methods are not supported.
switch statement	No	
break statement	No	
continue statement	No	
checked statement	No	
unchecked statement	No	
using statement	No	
lock statement	No	
is operator	No	
as operator	No	
"? :" ternary operator	No	
bitwise not operator	No	

- Subroutines and functions are parsed and added to the class **Global**.
- **DIMENSION** statements are converted to variable declarations with array type. The automatic typename is inferred from the first character of the declaration name.
- **DATA** statements are converted to a series of assign statements with the specified values.
- Line labels are converted to labels of the form "_<number>" where *number* is the line label.
- **DO** statements are converted to iteration statements.
- **GOTO** statements are converted to jump statements.
- **STOP** statements are converted to the method call `Environment.Exit(0);`.

A sample Fortran source code and its generated flowcharts can be found in appendix E.

5. CONCLUSIONS

In this thesis we developed a generic framework called SourceCodeParser which can be used to define the grammar of a language in a format called Grammar Definition Markup Language (GDML) which is based on XML. The framework can generate a tokenizer and parser for the language. The tokenizer can convert source code into a collection of tokens. The parser applies the grammar rules defined in the GDML to execute user defined transformations from tokens to higher level constructs. We used BUILD.NET classes as the target data structures and we successfully converted source codes to the flowchart representation.

Furthermore, the framework is not limited to target BUILD.NET classes. The framework can be used to target any other set of data structures to execute grammar rules for another purpose.

The main contributions of this thesis are the generic tokenizer and parser engines. The parser engine is stateful and executes greedily with backtracking capability. We can flexibly express the grammar rules that operate on a target set of data structures. We are only limited on the usage of the target data structures, but we can easily develop and target new data structures as well.

In the task of generating BUILD.NET flowcharts we have the limitation of the target data structures of BUILD.NET. For example BUILD.NET does not support a “switch” statement, so we must emulate it using “if” statements. We may lose some information when converting source code to BUILD.NET structures. We can emulate such features using equivalent supported features.

We tested SourceCodeParser by implementing parsers for the C# and Fortran languages. The issue is that unsupported features are either emulated or ignored. We can enhance the supported features by extending BUILD.NET.

A future work can be to extend the supported features of BUILD.NET, which allows higher quality results in the parsing process. Another possibility is to customize the code generation process to support more features than Microsoft CodeDOM supports. Another useful feature can be to improve the parser and tokenizer to continue the process if an error occurs and no match can be found at a specific input position.

APPENDIX A: GDML DESERIALIZING CLASSES

Once we have a GDML definition file, **SourceCodeParser** can generate the parser. This is done in two steps. First we deserialize the GDML content and load it as an instance of the **Grammar** class. Then we generate several parser classes. We map the GDML to the classes in **SourceCodeParser**. Now we will examine these classes.

A.1. XmlBaseDefinition Class

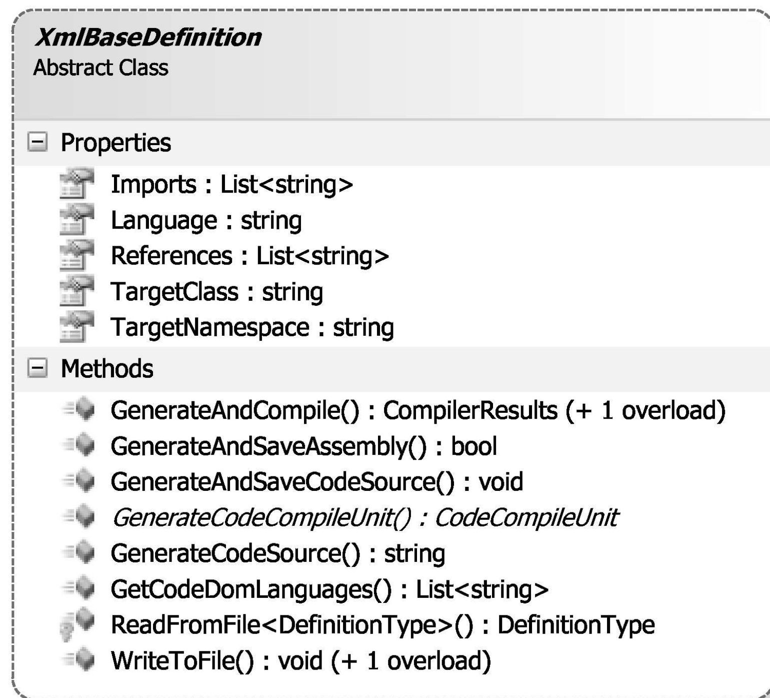


Figure A.1. XmlBaseDefinition class

The **XmlBaseDefinition** is the abstract base class for XML-serializable definitions targeting an assembly generation process. It contains fields corresponding to the common sections of such XML-serializable definitions and support for generating source code and assembly from that definitions.

- `string TargetNamespace`

  The attribute **targetNamespace** is deserialized to this property. This is the tar-

get namespace of generated code. All generated classes are put into this namespace.

- `string TargetClass`
  The attribute **targetClass** is deserialized to this property. This is the target main class of the generated code.
- `string Language`
  The attribute **language** is deserialized to this property. This is the programming language used in the code snippets in definitions. The code is generated in this programming language.
- `List<string> References`
  The XML section **References** is deserialized to this collection property. This is the list of assembly references for the generated code.
- `List<string> Imports`
  The XML section **Imports** is deserialized to this collection property. This is the list of namespace imports for the generated code.

XmlBaseDefinition class has methods for supporting:

- Object deserialization from XML files
- Object serialization to XML files
- Generation of source code and assembly from definitions
- GUI definition editors

The most important methods of the **XmlBaseDefinition** class are as follows:

- protected static DefinitionType ReadFromFile<DefinitionType>(
  string inputFile,
  string xmlRootNamespace,
  string xmlSchema)
  This static generic method reads an XML file and returns it as a deserialized **DefinitionType** object, where **DefinitionType** is a type extending **XmlBaseDefinition**. The parameter *inputFile* is the name of the input XML file. The pa-

rameter *xmlRootNamespace* is the root namespace of the XSD [26] schema of the XML file. The parameter *xmlSchema* is the string containing the XSD schema of the XML file.

- public void WriteToFile(string outputFile)
  This method serializes the definition to the specified XML file. The parameter *outputFile* is the name of the output XML file.
- public abstract CodeCompileUnit GenerateCodeCompileUnit()
  This abstract method is to be implemented in the classes that extend **XmlBaseDefinition**. This method generates the code and returns it as a **CodeCompileUnit** [27] object. The returned object can be used for further generating the source code or compiling the generated code as an assembly.
- public abstract CompilerResults GenerateAndCompile(
  string outputAssemblyFile,
  out string codeSource);
  This abstract method is to be implemented in the classes that extend **XmlBaseDefinition**. This method generates code from the definition and compiles it as an assembly. The parameter *outputAssemblyFile* is Name of the output assembly file. The parameter *codeSource* returns the generated source code. The method returns a **CompilerResults** objects which contains the compilation errors if there are any errors and the compiled assembly if there are no errors. If the parameter *outputAssemblyFile* is null or an empty string, the assembly is generated as an in-memory assembly [28] and it is not saved to a file.
- public string GenerateCodeSource()
  This method generates the code from the definition and returns it as a string.
- public void GenerateAndSaveCodeSource(string sourceFile)
  This method generates the code from the definition and saves the generated source code to a file. The parameter *sourceFile* is the name of the output source code file.
- public bool GenerateAndSaveAssembly(string outputAssemblyFile)
  This method generates the code from the definition and compiles it as an assembly to a file. The parameter *outputAssemblyFile* is the name of the output assembly file. The method returns true on success.

- public static List<string> GetCodeDomLanguages()

  This static method returns the list of known CodeDom languages in the system. This method supports the GUI definition editors. The return value is the list of valid values for the **Language** property of the **XmlBaseDefinition**.

A.2. Grammar Class

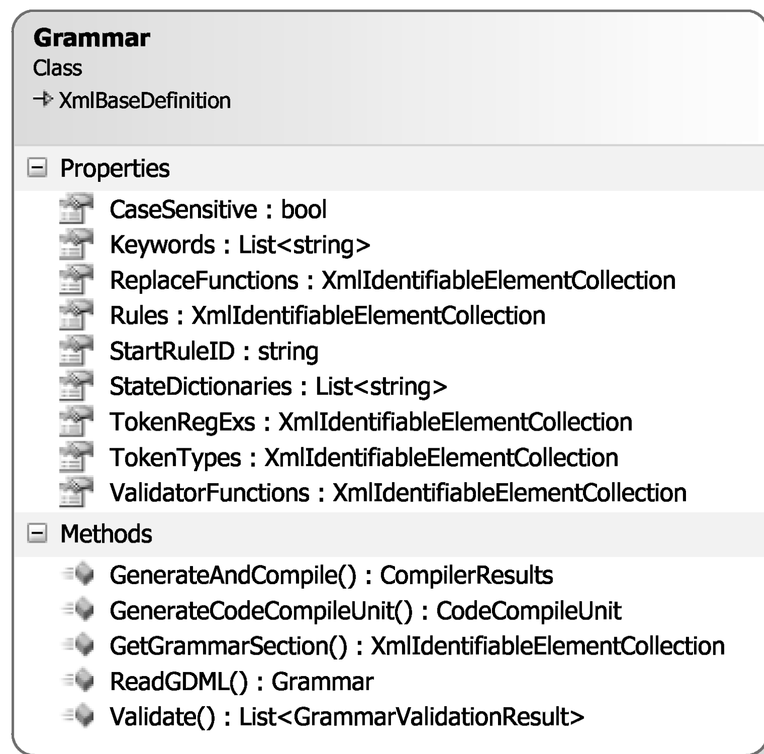


Figure A.2. Grammar class

The **Grammar** class corresponds to a GDML definition. The **Grammar** class extend the class **XmlBaseDefinition** and contains properties corresponding to the GDML specific sections and attributes:

- `string StartRuleID`

  The attribute **startRuleID** is deserialized to this property. This is the identifier of the start rule that will be executed in the grammar parse phase.

- `bool CaseSensitive`

  The attribute **caseSensitive** is deserialized to this property. This boolean flag is used to specify that the keywords in the language are case-sensitive.

- `List<string> Keywords`

  The GDML section **Keywords** is deserialized to this collection property. This is the list of keywords in the language being parsed.

- `XmlIdentifiableElementCollection TokenRegExs`

  The GDML section **TokenRegExs** is deserialized to this collection property. **RegEx** elements are loaded as **XmlTokenRegEx** objects to the collection.

- `XmlIdentifiableElementCollection TokenTypes`

  The GDML section **TokenTypes** is deserialized to this collection property. **TokenType** elements are loaded as **XmlTokenType** objects to the collection.

- `List<string> StateDictionaries`

  The GDML section **StateDictionaries** is deserialized to this collection property. This is the list of names of parser state dictionaries.

- `XmlIdentifiableElementCollection ValidatorFunctions`

  The GDML section **ValidatorFunctions** is deserialized to this collection property. **Function** elements are loaded as **XmlIdentifiableFunction** objects to the collection.

- `XmlIdentifiableElementCollection ReplaceFunctions`

  The GDML section **ReplaceFunctions** is deserialized to this collection property. **Function** elements are loaded as **XmlIdentifiableFunction** objects to the collection.

- `XmlIdentifiableElementCollection Rules`

  The GDML section **Rules** is deserialized to this collection property. **Rule** elements are loaded as **MatchRule** objects to the collection.

Grammar class has methods for supporting:

- Grammar deserialization from files
- Grammar serialization to files
- Generation of parser code and assembly
- GDML GUI editor
- Grammar validation

The most important methods of the **Grammar** class are as follows:

- public static Grammar ReadGDML(string grammarFile)
  This static method reads a GDML file and returns it as a **Grammar** object. The parameter *grammarFile* is the name of the input GDML file.
- public override CodeCompileUnit GenerateCodeCompileUnit()
  This method implements the corresponding abstract method in the base class.
- public CompilerResults GenerateAndCompile(
  string outputAssemblyFile,
  out string codeSource)
  This method implements the corresponding abstract method in the base class.
- public XmlIdentifiableElementCollection GetGrammarSection(
  GrammarSection section)
  This method returns the collection of items of a given section of the grammar. The parameter *section* is the grammar section whose collection is to be returned.
- public List<GrammarValidationResult> Validate(bool validateCodeSnippets)
  This method validates the grammar against grammar constraints and returns the list of errors and warnings as a list of **GrammarValidationResult** objects. The parameter *validateCodeSnippets* is a boolean flag to specify to validate all code snippets by compiling each of them. **GrammarValidationResult** contains the type (error or warning), message, the GDML section and the corresponding item in the section.

A.3. GrammarSection Enumeration

This enumeration represents editable sections in a GDML definition. The enumeration has the following fields:

- **Grammar:** This field represents the grammar object itself.
- **References:** This field represents the collection **References** of the grammar.
- **Imports:** This field represents the collection **Imports** of the grammar.
- **StateDictionaries:** This field represents the collection **StateDictionaries** of

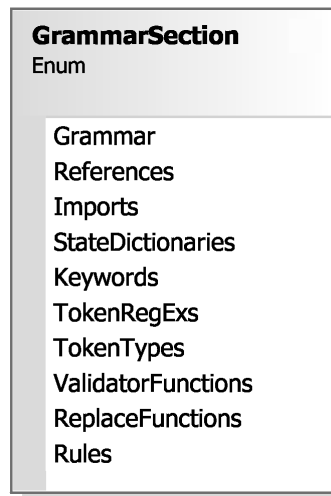


Figure A.3. GrammarSection enumeration

the grammar.

- **Keywords:** This field represents the collection **Keywords** of the grammar.
- **TokenRegExs:** This field represents the collection **TokenRegExs** of the grammar.
- **TokenTypes:** This field represents the collection **TokenTypes** of the grammar.
- **ValidatorFunctions:** This field represents the collection **ValidatorFunctions** of the grammar.
- **ReplaceFunctions:** This field represents the collection **ReplaceFunctions** of the grammar.
- **Rules:** This field represents the collection **Rules** of the grammar.

A.4. MatchQuantity Enumeration

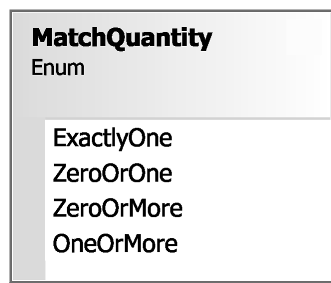


Figure A.4. MatchQuantity enumeration

This enumeration represents the quantity of items to match in a parse rule. The

enumeration has the following fields:

- **ExactlyOne:** Match succeeds if exactly one object is matched. This field is serialized to the XML with the value “1”.
- **ZeroOrOne:** Match succeeds if zero or one object is matched. This field is serialized to the XML with the value “?”.
- **ZeroOrMore:** Match succeeds whether or not any number of objects match. This field is serialized to the XML with the value “*”.
- **OneOrMore:** Match succeeds if any positive number of objects match. This field is serialized to the XML with the value “+”.

A.5. MatchType Enumeration

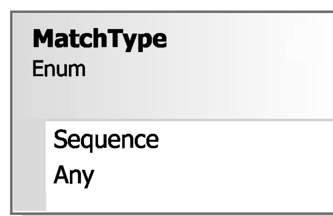


Figure A.5. MatchType enumeration

This enumeration represents the type of the matching algorithm of a rule. The enumeration has the following fields:

- **Sequence:** The rule matches a sequence of rule items. This field is serialized to the XML with the value “sequence”.
- **Any:** The rule matches any one of the rule items contained in the rule. This field is serialized to the XML with the value “any”.

A.6. XmlIdentifiableElement Class

This abstract class is the base class for serializable XML elements identifiable by an attribute named **ID**. The element identifiers must be unique in their XML sections. The class implements the interface **IComparable** for comparing the objects by their identifier alphabetically.

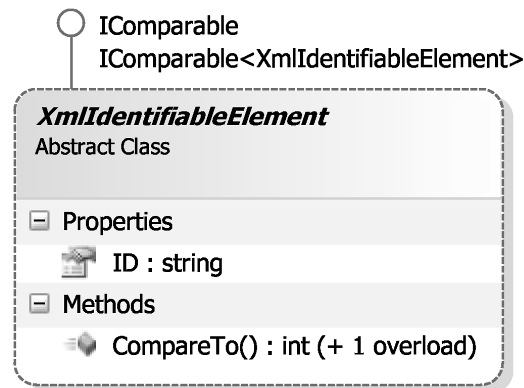


Figure A.6. XmlIdentifiableElement class

The class has the following properties:

- `string ID`: This is the unique identifier of the item. This property is mapped to the XML attribute named **ID**.

The class has the following methods:

- public int CompareTo(object obj)
  This method implements the interface **IComparable** and is used to compare two instances of objects for ordering by their unique identifiers.

A.7. XmlIdentifiableElementCollection Class

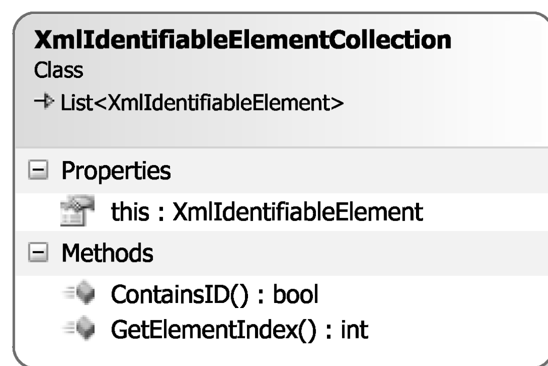


Figure A.7. XmlIdentifiableElementCollection class

This is a collection of objects with base type **XmlIdentifiableElement**. The following GDML sections are mapped to grammar properties of type **XmlIdentifi-**

ableElementCollection:

- **TokenRegExs**
- **TokenTypes**
- **ValidatorFunctions**
- **ReplaceFunctions**
- **Rules**

This class has the following methods:

- public int GetElementIndex(string id)
  This method returns the zero-based index of the first occurrence of the specified identifier value in the collection. The parameter *id* is the identifier of the element to look for. The return value will be -1 if the specified identifier is not found in the collection.
- public bool ContainsID(string id)
  This method returns true if the specified identifier exists in the collection. The parameter *id* is the identifier of the element to look for.
- XmlIdentifiableElement this[string id]
  This indexer method returns the items in the collection by their identifier instead of their index in the collection. The parameter *id* is the identifier of the element to look for. The return value will be "null" if the specified identifier is not found in the collection.

A.8. XmlIdentifiableReference Class

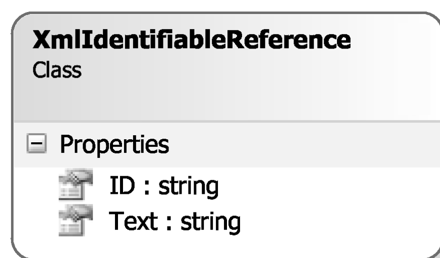


Figure A.8. XmlIdentifiableReference class

This class is used for serializable XML elements that refer to an element identifiable by an attribute named "ID" or that may contain text content. It has the following properties:

- `string ID`: This is the unique identifier of the item being referenced. This property is mapped to the XML attribute named **ID**.
- `string Text`: This is the text content of the element. This property is mapped to the XML text content of the XML element.

A.9. XmlTokenRegEx Class

Figure A.9. XmlTokenRegEx class

This class is used to deserialize token regular expressions defined in the grammar section **TokenRegExs**. The base class of this class is **XmlIdentifiableElement**. **RegEx** elements in that section are mapped to instances of this class.

This class has a collection member named **Parts**, which is mapped to the **Parts** element. **Part** elements are loaded as **XmlIdentifiableReference** objects to this collection. Each regular expression part may contain either text content or a reference to another token regular expression. The value of a token regular expression is the concatenation of the values of referred token regular expressions and the text contents.

A.10. XmlTokenType Class

This class is used to deserialize token type definitions in the grammar section **TokenTypes**. The base class of this class is **XmlIdentifiableElement**. **TokenType** elements in that section are mapped to instances of this class.

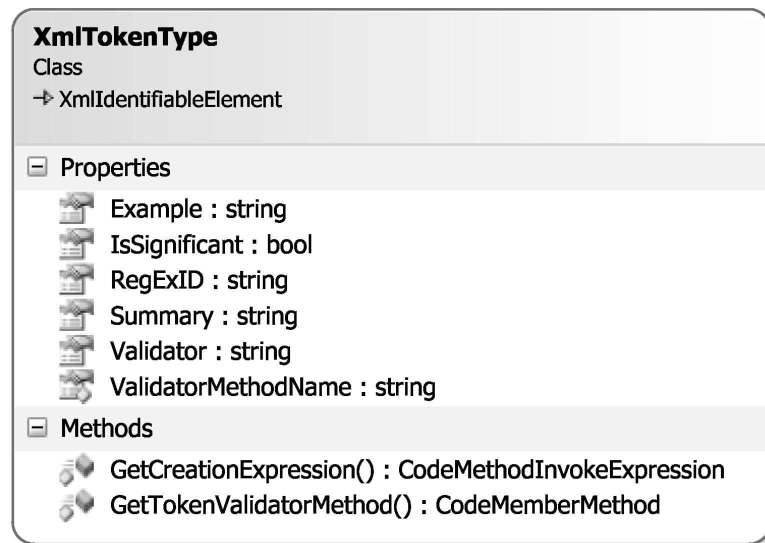


Figure A.10. XmlTokenType class

The class has the following properties:

- `string RegExID`: The attribute **RegExID** is deserialized to this property. This is the identifier of the token regular expression from the section **TokenRegExs** that matches the specified token type.
- `bool IsSignificant`: The attribute **IsSignificant** is deserialized to this property. The default value is "true". This property is a flag to specify whether this token type is significant in parsing. Insignificant token types are ignore during parsing process.
- `string Summary`: The optional element **Summary** is deserialized to this property. This property can be used to describe the token type.
- `string Example`: The optional element **Example** is deserialized to this property. This property can be used to give an example of the token type.
- `string Validator`: The optional element **Validator** is deserialized to this property. This property contains a code snippet with the prototype **TokenValidatorDelegate**.
- `string ValidatorMethodName`: This property returns the name of the generated method for the token validator in the generated code parser class. The return value is of the form "TokenValidator_<id>" where "id" is the identifier of the token type.

The class has the following methods:

- internal protected CodeMemberMethod GetTokenValidatorMethod()
  This method returns the generated token validator function. It will be used when generating the code parser from the grammar.
- internal protected CodeMethodInvokeExpression GetCreationExpression()
  This method returns the expression that creates the corresponding token type definition in the generated parser. It will be used to create the token type in the constructor of the generated code parser.

A.11. XmlIdentifiableFunction Class

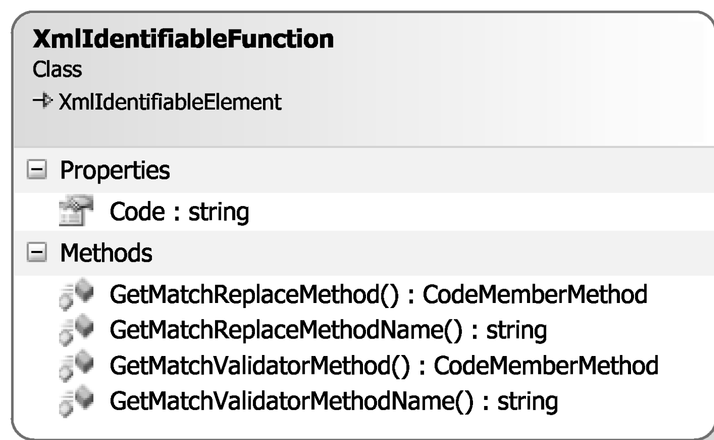


Figure A.11. XmlIdentifiableFunction class

This class is used to deserialize functions defined in the grammar sections **ValidatorFunctions** and **ReplaceFunctions**. The base class of this class is **XmlIdentifiableElement**.

The class has the following properties:

- `string Code`: This property maps to the XML text content of the **Code** element and contains the code snippet of the function. The language of the code snippet is specified in the grammar attribute **language**.

The class has the following methods:

- protected internal CodeMemberMethod GetMatchValidatorMethod()
  This method returns the generated reusable match validator function. It will be used when generating the code parser from the grammar.
- protected internal static string GetMatchValidatorMethodName(XmlIdentifiableReference reference)
  This static method returns the name of a referred method for a reusable match validator function in the generated code parser class. The parameter *reference* is the reference to the validator function. The return value is of the form "MatchValidator_<id>" where "id" is the identifier of the reference.
- protected internal CodeMemberMethod GetMatchReplaceMethod()
  This method returns the generated match replace function. It will be used when generating the code parser from the grammar.
- protected internal static string GetMatchReplaceMethodName(XmlIdentifiableReference reference)
  This static method returns the name of a referred method for a reusable match replace function in the generated code parser class. The parameter *reference* is the reference to the replace function. The return value is of the form "MatchReplace_<id>" where "id" is the identifier of the reference.

A.12. MatchRule Class

This class is used to deserialize and execute the grammar rules in the section **Rules**. The base class of this class is **XmlIdentifiableElement**.

The class has the following properties:

- `string Summary`: This property is mapped to the optional element **Summary** that describes the rule.
- `MatchType MatchType`: This property is mapped to the attribute **matchType**. It has the default value `MatchType.Sequence` and it represents the type of the matching algorithm of the rule.
- `List<MatchRuleItem> Match`: The collection in the **Match** element is loaded

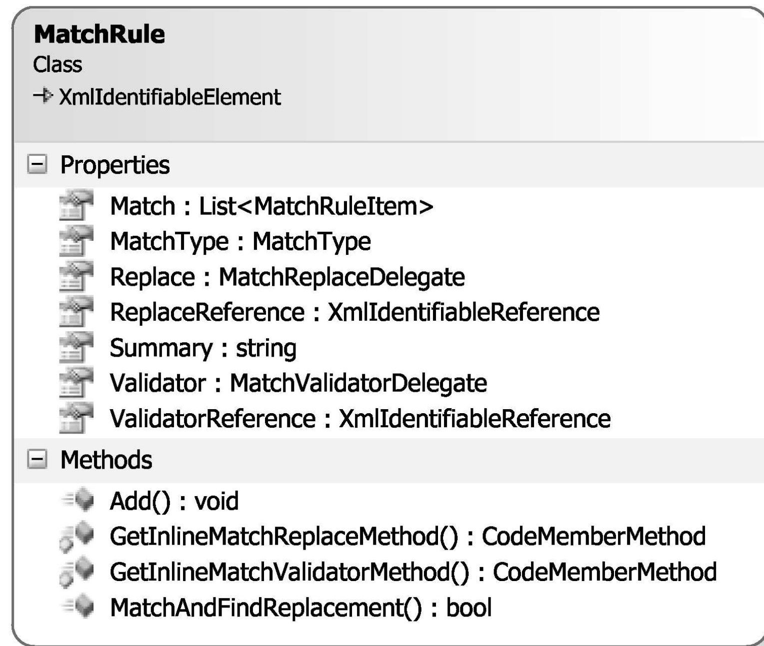


Figure A.12. MatchRule class

to this property. **Token** elements are loaded as **MatchRuleItemToken** objects and **Recursive** elements are loaded as **MatchRuleItemRecursive** objects to the collection. This is the list of items to match with the token collection, such that the rule matches.

- `XmlIdentifiableReference ValidatorReference`: This property is mapped with the optional element **Validator**. This property may refer a validator function from the section **ValidatorFunctions** or define a custom inline function with the prototype **MatchValidatorDelegate** shown in figure 3.12.
- `XmlIdentifiableReference ReplaceReference`: This property is mapped with the element **Replace**. This property may refer a replace function from the section **ReplaceFunctions** or define a custom inline function with the prototype **MatchReplaceDelegate** shown in figure 3.14.
- `MatchValidatorDelegate Validator`: This property is used at run-time to store the actual delegate referred by the property **ValidatorReference**.
- `MatchReplaceDelegate Replace`: This property is used at run-time to store the actual delegate referred by the property **ReplaceReference**.

The class has the following methods:

- protected internal CodeMemberMethod GetInlineMatchValidatorMethod()
  This method returns the generated inline match validator function. It will be used when generating the code parser from the grammar.
- protected internal CodeMemberMethod GetInlineMatchReplaceMethod()
  This method returns the generated inline match replace function. It will be used when generating the code parser from the grammar.
- public void Add(MatchRuleItem item)
  This method adds a new rule item to the **Match** collection. This method is called by the constructor of the generated parser class to add the rule items to the rule. The parameter *item* is the rule item to be added.
- public bool MatchAndFindReplacement(TokenCollection tokens, int index, ParserState state, out object replacement, out int newIndex, out ParserStateDifference stateDifference)
  This method first executes the rule's match part. If the match succeeds and validates successfully the replace part is executed. The parameter *tokens* is the input token collection. The parameter *index* is the first element in the input collection to check for the match. The parameter *state* is the state of the parsing process. The parameter *replacement* returns the replacement object for the rule. The parameter *newIndex* returns the next index for applying a new rule in the input collection for a successful match. The parameter *stateDifference* is the parser state difference after the replace method call. The parameter *newIndex* will have the value "-1" if the match fails. The parameter stateDifference will be "null" if the match fails or the state difference is empty. The return value is "true" if the rule matches at the specified index of the token collection.

A.13. MatchRuleItem Class

This abstract class is the base class for classes used to deserialize and execute match rule items. These classes are **MatchRuleItemToken** and **MatchRuleItem-Recursive**. A match rule item represents a series of tokens of uniform type or a series of rule matches for a recursive rule reference.

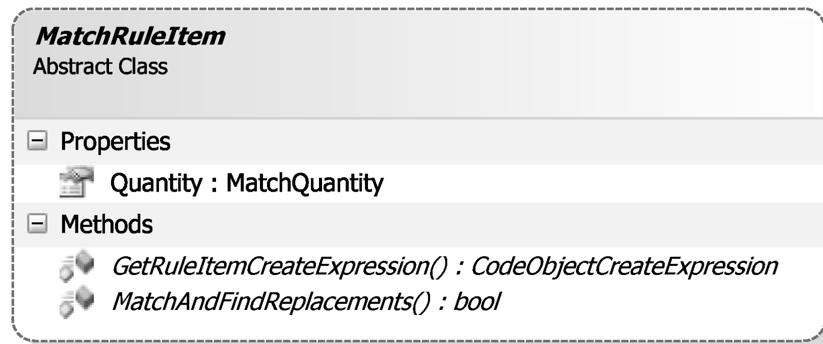


Figure A.13. MatchRuleItem class

The class has the following properties:

- `MatchQuantity Quantity`: This is the quantity specifier for matching the tokens or executing recursive rules.

The class has the following methods:

- protected internal abstract CodeObjectCreateExpression GetRuleItemCreateExpression();
  This method is to be implemented in classes that extend **MatchRuleItem**. It returns the object create expression that will initialize the match rule item in the generated code parser constructor.
- protected internal abstract bool MatchAndFindReplacements(
  TokenCollection tokens, int index,
  ParserState state, MatchResultSlot resultSlot)
  This method is to be implemented in classes that extend **MatchRuleItem**. This method executes the matching logic of the rule item on a token collection. The parameter *tokens* is the input token list. The parameter *index* is the first index in *tokens* to look for a match. The parameter *state* is the current parser state object. The matched objects are stored in the parameter *resultSlot*. The quantity specifier of the rule item is used to execute a series of matches.

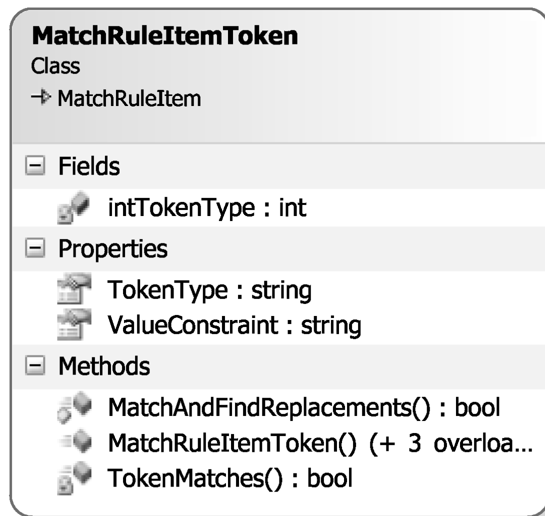


Figure A.14. MatchRuleItemToken class

A.14. MatchRuleItemToken Class

This class is used to deserialize and execute rule items with element name **Token**. The base class of this class is **MatchRuleItem**.

The class has the following properties and fields:

- `private int intTokenType`: This field stores at run-time the generated integer enumeration constant for the token type.
- `string TokenType`: This property is mapped to the XML attribute **tokenType**. It specifies the name of the token type to match. At run-time this property is used to set the value of the field **intTokenType**.
- `string ValueConstraint`: This optional property is mapped to the XML attribute **token**. If this property contains a non-empty string value, then it is used as a filter for the token value to match by the rule item.

The class has the following methods:

- private bool TokenMatches(Token token, bool caseSensitive)
  This method checks whether a token matches the type condition and the optional

value constraint. The parameter *token* is a token to look for a match by this rule item. The parameter *caseSensitive* is a flag to specify case sensitive matching of token values.

- protected internal override bool MatchAndFindReplacements(
  TokenCollection tokens, int index,
  MatchResultSlot resultSlot)
  This method implements the abstract method in the base class. This method looks for a series of token matches in the token list according to the quantity, token type and optional token value constraints. The parameter *tokens* is the input token list. The parameter *index* is the first index in *tokens* to look for a match. The parameter *resultSlot* is the target match result slot for storing the matched tokens. The method returns true on a successful match.

A.15. MatchRuleItemRecursive Class

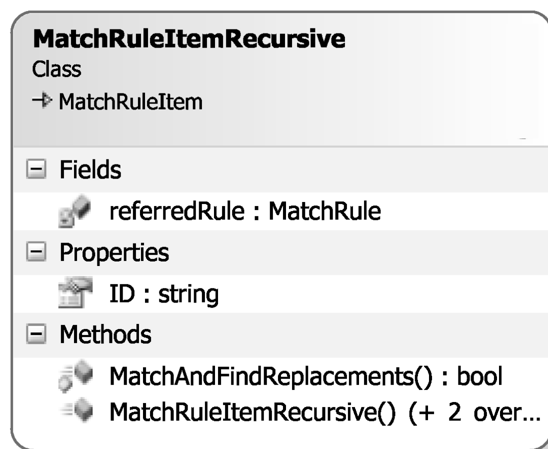


Figure A.15. MatchRuleItemRecursive class

This class is used to deserialize and execute rule items with element name **Recursive**. The base class of this class is **MatchRuleItem**.

The class has the following properties and fields:

- `string ID`: This property is mapped to the XML attribute **ID**. It is the identifier of the rule being referred.

- `private MatchRule referredRule`: This field is used at the run-time to store the actual rule instance being referred.

The class has the following methods:

- public bool MatchAndFindReplacements(TokenCollection tokens, int index, ParserState state, MatchResultSlot resultSlot)
  This method executes the referred rule's match part. If the match succeeds the replace part is executed. The parameter *tokens* is the input token list. The parameter *index* is the first index in *tokens* to look for a match. The parameter *state* is the current parser state object. The replacement results are stored in the parameter *resultSlot*. The quantity specifier of the rule item is used to match a series of the referred rule.

APPENDIX B: UTILITY CLASSES

This section contains utility classes used in the **SourceCodeParser** framework.

B.1. ParseException Class

Figure B.1. ParseException class

This class is an exception class that will be thrown in the tokenizing and parsing processes. This class inherits the class `System.Exception` which is the base class for .NET runtime exceptions.

B.2. GrammarValidationResult Class

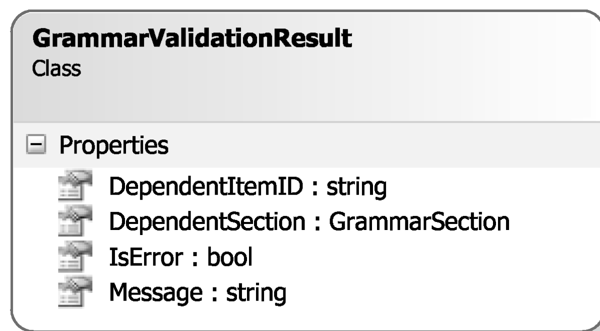


Figure B.2. GrammarValidationResult class

This class represents the result of a single failure in the validation of the grammar. The failure can be a warning or an error. This class is used in the grammar validation process in the GDML GUI Editor and has the following properties:

- `GrammarSection DependentSection`: The grammar section that the validated item belongs to.
- `string DependentItemID`: The identifier of the validated item under the grammar section.

- `bool IsError`: Boolean flag to specify whether the validation result is an error or a warning.
- `string Message`: The error message to be displayed.

B.3. XmlProjectDefinition Class

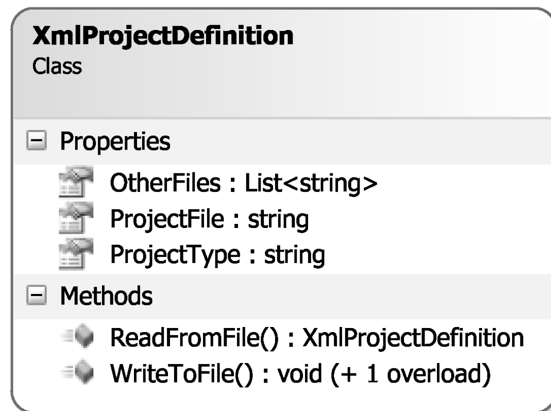


Figure B.3. XmlProjectDefinition class

This class is an XML-serializable class for storing a project composed of a main project file and related list of files to be used on execution of the main project file. This class is used for handling projects in the GUI editors. The serialized XML for this class has the root element **Project**.

The class the following properties:

- `string ProjectType`: This is the type identifier for the project. This property is serialized as the XML element **ProjectType**.
- `string ProjectFile`: Filename of the main project file. This property is serialized as the XML element **ProjectFile**.
- `List<string> OtherFiles`: This collection property contains the list of files used for running the main project. This property is serialized as a collection with element name **OtherFiles**. Each item in the collection is serialized with the element name **File**.

The class has the following methods:

- public static XmlProjectDefinition ReadFromFile(string inputFile)

  This method reads an input file and returns it as a deserialized project object. The parameter *inputFile* is the name of the input file.

- public void WriteToFile(string outputFile)

  This method writes the definition XML to the specified file. The parameter *outputFile* is the name of the output file.

APPENDIX C: TOKENIZER AND PARSER CLASSES

C.1. TokenDefinition Class

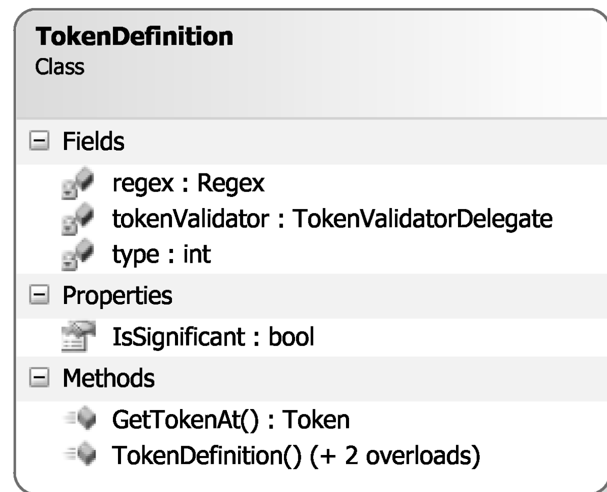


Figure C.1. TokenDefinition class

This class contains the regular expression necessary for the tokenizing process.

The class has the following properties and fields:

- `private int type`: This field is the enumeration constant for the token type to be parsed.
- `private Regex regex`: This field is the compiled regular expression object that matches the token.
- `bool IsSignificant`: This property is a flag to specify whether this token type is significant in parsing.
- `private TokenValidatorDelegate tokenValidator`: This field is the optional delegate function to call to validate a token after the regular expression matches.

The class has the following methods:

- public Token GetTokenAt(ref string buffer, ref int offset)
  This method returns the matching string if the token type matches and validates

at a specific position in a string. The parameter *buffer* is the string to look into for a match. The parameter *offset* is the character index inside *buffer* to look for the token. The method returns the token string if the token exists at position *offset* in *buffer* or "null" if the token does not exist at that position.

C.2. Token Class

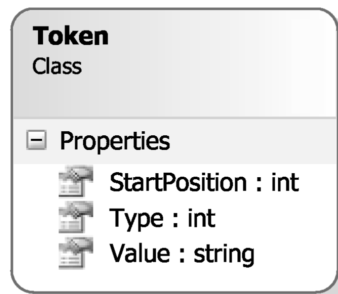


Figure C.2. Token Class

This class contains the information for a single token found after the tokenizing process. It has the following properties:

- `int StartPosition`: This is the starting character position of the token inside the source.
- `int Type`: This is the type of the token. This field has the value of one of the constants generated for the token types.
- `string Value`: This is the string value of the token.

C.3. TokenCollection Class

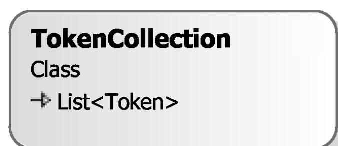


Figure C.3. TokenCollection class

This is a collection of **Token** objects. This class represents a collection of tokens found after the tokenizing process. **TokenCollection** object is the input of the parsing process.

C.4. BaseCodeParser Class

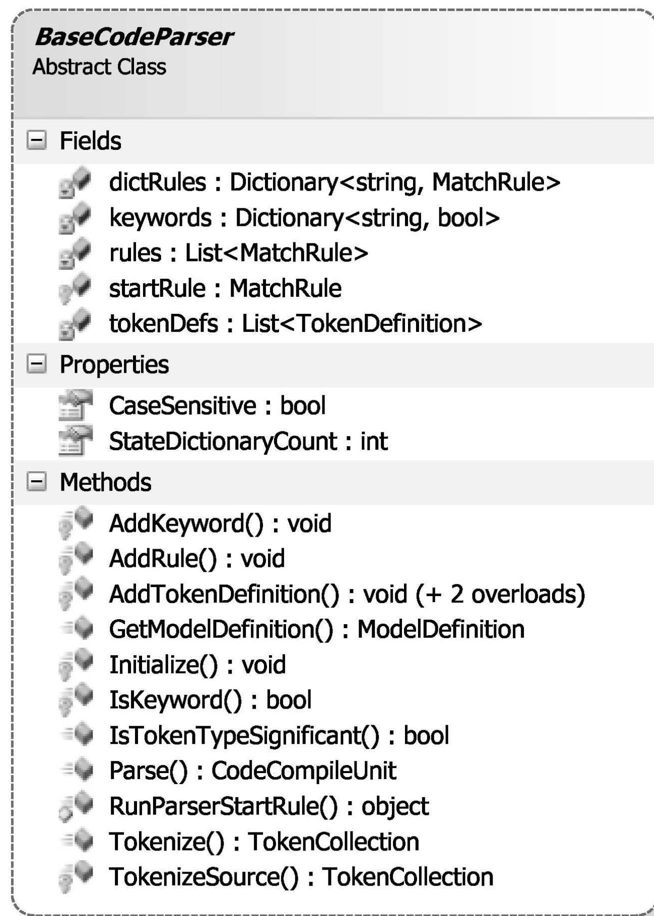


Figure C.4. BaseCodeParser class

This class is the base class of the generated code parser class. It has support for the tokenizing and parsing phases.

The class has the following properties and fields:

- `private Dictionary<string, bool> keywords`
  This dictionary field is a lookup table that contains language specific keywords.
- `private List<TokenDefinition> tokenDefs`: This field is the list of token definitions to find language specific tokens.
- `private Dictionary<string, MatchRule> dictRules`: This field is the dictionary of rules indexed by their identifier.
- `private List<MatchRule> rules`: This field is the list of grammar rules of the

language.

- `protected MatchRule startRule`: This field is the start rule that will be executed in the grammar parse phase.
- `bool CaseSensitive`: This property is a flag to specify that the keywords in the language are case-sensitive.
- `int StateDictionaryCount`: This property is the number of parser state dictionaries defined in the grammar.

The class has the following methods:

- protected void Initialize(bool caseSensitive, int keywordCount, int tokenDefinitionCount, int stateDictionaryCount, int matchRuleCount)
  This method initializes the parser object by setting the capacities of collections of the keywords, token definitions, parser state dictionaries and grammar rules. The parameter *caseSensitive* is a flag to specify that the keywords in the language are case-sensitive. The parameter *keywordCount* is the number of keywords. The parameter *tokenDefinitionCount* is the number of token definitions. The parameter *stateDictionaryCount* is the number of the parser state dictionaries in a parser state for this language. The parameter *matchRuleCount* is the number of grammar rules in this language.
- protected void AddKeyword(string keyword)
  This method adds a new keyword to the list of keywords. The parameter *keyword* is the keyword text. If the keywords are case insensitive, then they are saved in lower case inside the dictionary field **keywords**. So we can ignore the case in the lookup. This method is to be called by the constructor of the generated code parser.
- protected bool IsKeyword(string keyword)
  This method checks whether a string is a language specific keyword. The parameter *keyword* is the string to check for. This method can be called in token validator functions to distinguish keywords and identifiers.
- protected void AddTokenDefinition(int type, string pattern, bool isSignificant, TokenValidatorDelegate tokenValidator)

This method adds a new token definition to the list of token definitions. The parameter *type* is the enumeration constant of the token type. The parameter *pattern* is the regular expression that matches the token. The parameter *isSignificant* is an optional flag with the default value "true" to specify whether the token is significant in parsing. The parameter *tokenValidator* is the optional delegate function to call to validate a token after the regular expression matches.

- protected void AddRule(MatchRule rule)

  This method adds a new rule to the list of grammar rules. The parameter *rule* is the rule to be added to the list.

- public bool IsTokenTypeSignificant(int tokenType)

  This method checks whether the specified token type is significant in parsing. The parameter *tokenType* is the predefined integer constant for the token type.

- protected internal object RunParserStartRule(

  TextReader codeStream, out List<object> logItems)

  This method executes the parser engine by executing the start rule and returns the replacement object of the rule. The parameter *codeStream* is the input stream to parse. The parameter *logItems* returns the log items stored by the parser.

- protected TokenCollection TokenizeSource(string source)

  This method parses an input source for low level language tokens. The parameter *source* is the source code to parse. The method returns a **TokenCollection** object for the list of parsed tokens or "null" if an error occurs.

- public TokenCollection Tokenize(TextReader codeStream)

  This method parses an input source stream for low level language tokens. The parameter *codeStream* is the input stream to parse. The method returns a **TokenCollection** object for the list of parsed tokens or "null" if an error occurs.

- public ModelDefinition GetModelDefinition(TextReader codeStream)

  This method executes the parser engine by executing the start rule and returns a BUILD.NET model definition. The parameter *codeStream* is the input stream to parse. The method returns a **ModelDefinition** object or "null" if an error occurs.

- public CodeCompileUnit Parse(TextReader codeStream)

  This method implements returns a **CodeCompilationUnit** object after parsing

the input source.

C.5. ParserStateDictionary Class

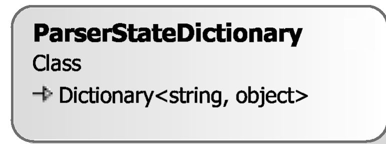


Figure C.5. ParserStateDictionary class

This class is a dictionary of objects used as a state storage in the parsing process. The keys in the dictionary are of type **string** and the values can be of any type. Typical usage of this class is to store and lookup a category of state information in the parsing process.

C.6. ParserState Class

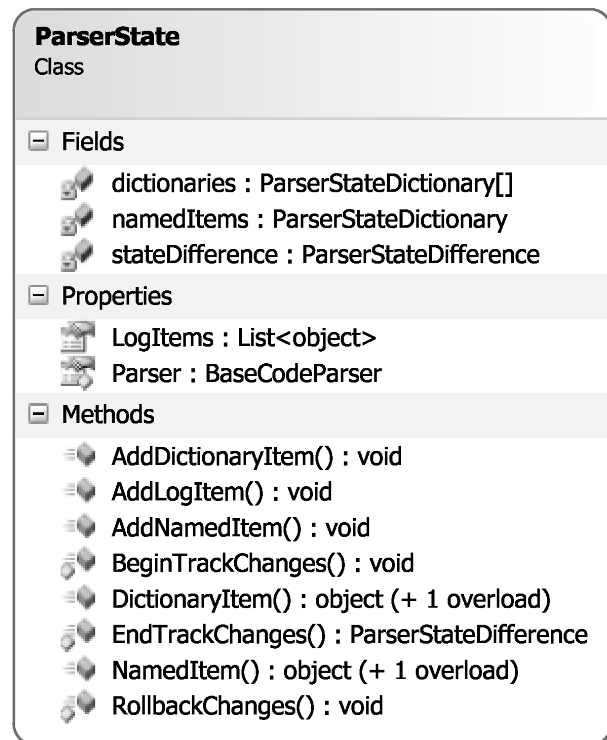


Figure C.6. ParserState class

This class is used for storing user defined state of the parsing process. The parser state consists of dictionary items, named items and log items.

The parser state contains a predefined set of custom dictionaries of type **ParserStateDictionary** for dictionary items. The dictionaries are defined in the GDML section **ParserStateDictionaries**. There is also another **ParserStateDictionary** for storing named items in the parser state. Also we have a collection of user defined log items.

The class contains methods for adding new items and reading items, which can be used in rule validator and rule replace functions. Note that there are no methods to remove an item explicitly. The state changes applied by the items of a rule are automatically rolled back after the rule's match part executes. The changes applied by the replace function of a rule referred from a recursive rule item persist when executing next rule items in the rule.

The log items behave different than named items and dictionary items. The items added to the log collection are removed from the collection only if the rule fails and backtracks.

The class has the following fields and properties:

- `private ParserStateDictionary[] dictionaries`: This field is the predefined set of state dictionaries for dictionary items.
- `private ParserStateDictionary namedItems`: This field is the state dictionary for named items.
- `private ParserStateDifference stateDifference`: This field is the object used for tracking changes to the parser state. If this field is "null" the tracking is disabled. If this field has an instance the state changes are stored to this field. The state changes are the newly added dictionary items, named items and log items.
- `List<object> LogItems`: This property is the list of log items in the state.
- `protected internal BaseCodeParser Parser`:
  This property is the active parser object that has created the parser state object.

The class has the following methods:

- protected internal void BeginTrackChanges()

  This method initializes the state change tracking by creating a new object on field **stateDifference**. This is done before a rule replacement method call.
- protected internal ParserStateDifference EndTrackChanges()

  This method ends the state change tracking and returns the tracked state difference as a **ParserStateDifference** object. This is done usually after a rule replacement method call. The return value will be "null" if there are no state changes.
- protected internal void RollbackChanges(ParserStateDifference stateDifference)

  This method removes the changes in the state difference from the parser state. The parameter *stateDifference* is the state difference to rollback.
- public void AddNamedItem(string key, object item)

  This method registers a named item in the parser state. The parameter *key* is the key of the item. The parameter *item* is the named item to be added to the state. If the specified key is already used an exception will be thrown.
- public object NamedItem(string key)

  This method returns a named item from the parser state. The parameter *key* is the key of the item. If the specified key is not found an exception will be thrown.
- public T NamedItem<T>(string key)

  This method returns a strongly typed named item from the parser state. The parameter *key* is the key of the item. If the specified key is not found an exception will be thrown. The parameter *key* is the key of the item. The type parameter *T* is the type of the object to return. If the specified key is not found or there is a type mismatch an exception will be thrown.
- public void AddDictionaryItem(int index, string key, object item)

  This method registers a dictionary item in the parser state. The parameter *index* is the index of the target dictionary in the parser state. The parameter *key* is the key of the item. The parameter *item* is the dictionary item to be added to the state. If the specified key is already used in the target dictionary an exception will be thrown.

- public object DictionaryItem(int index, string key)

  This method returns a dictionary item from the parser state. The parameter *index* is the index of the target dictionary in the parser state. The parameter *key* is the key of the item. The type parameter *T* is the type of the object to return. If the specified key is not found in the target dictionary an exception will be thrown.
- public T DictionaryItem<T>(int index, string key)

  This method returns a strongly typed dictionary item from the parser state. The parameter *index* is the index of the target dictionary in the parser state. The parameter *key* is the key of the item. If the specified key is not found in the target dictionary an exception will be thrown.
- public void AddLogItem(object item)

  This method registers a log item in the parser state. The parameter *item* is the item to be added to the log items.

C.7. IndexKeyPair Class

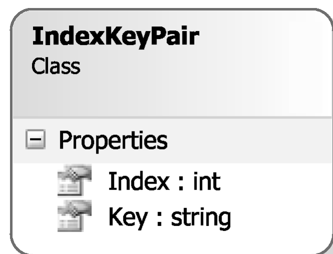


Figure C.7. IndexKeyPair class

This class represents a dictionary item added to a parser state dictionary with the specified key. It has the following properties:

- `int Index`: The index of the dictionary in the parser state
- `string Key`: Key of the item added to the dictionary

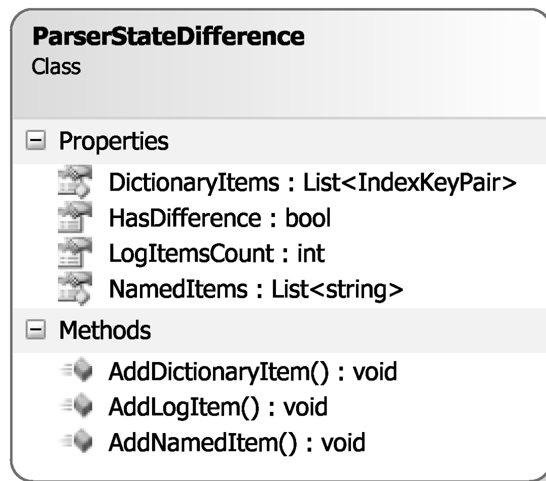


Figure C.8. ParserStateDifference class

C.8. ParserStateDifference Class

This class represents the difference of parser states in terms of newly added state objects. The state difference contains a list of keys for newly added named items and a list of **IndexKeyPair** objects for newly added dictionary items.

The **ParserStateDifference** object is used by the **ParserState** class for tracking the changes of the parser state. A **ParserStateDifference** object can be used to undo the changes that is applied to the parser state. That is done by removing the items from the **ParserState** object, that are stored in the **ParserStateDifference** object.

The class has the following properties:

- `protected internal List<IndexKeyPair> DictionaryItems`:
  This is the list of dictionary items in the state difference identified by a dictionary index and a key.
- `protected internal List<string> NamedItems`:
  This is the list of named items in the state difference identified by a key.
- `public int LogItemsCount`: This is the number of log items in the state difference.

The class has the following methods:

- public void AddDictionaryItem(int index, string key)
  This method registers a dictionary item in the state difference. The parameter *index* is the index of the dictionary in the parser state. The parameter *key* is the key of the item added to the dictionary.
- public void AddNamedItem(string key)
  This method registers a named item in the state difference. The parameter *key* is the key of the named item added to the dictionary.
- public void AddLogItem()
  This method registers a log item in the state difference.

C.9. MatchResultSlotItem Class

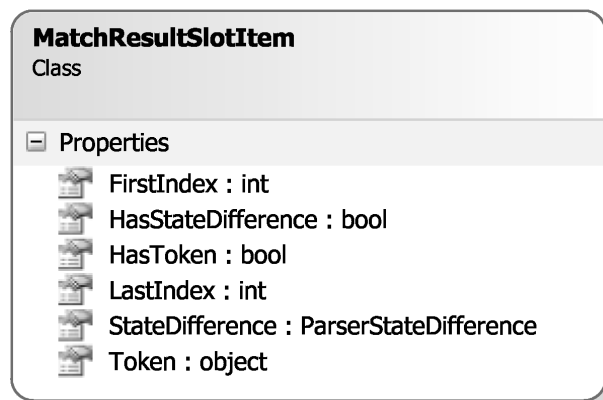


Figure C.9. MatchResultSlotItem class

This class represents a single object matched by a rule item. This object can be a simple token or a higher level construct returned from a recursive rule execution.

The class has the following properties:

- `object Token`: This is one object matched by the corresponding rule item.
- `int FirstIndex`: This is the first index of the token range in the original token collection that correspond to the matched object.
- `int LastIndex`: This is the last index of the token range in the original token

collection that correspond to the matched object. This value will be **FirstIndex-1** if the range is empty.

- `ParserStateDifference StateDifference`: This property is the parser state difference after the current item is matched in the parser. If the corresponding rule item is a recursive one that causes a change in the state, this property will hold the state changes. So we can rollback the changes if we backtrack a match on a **MatchResultSlot**. This property will be "null" if the state difference is empty.
- `public bool HasToken`: This property returns true if the result slot item has been set a non-null matched object.
- `public bool HasStateDifference`: This method returns true if the result slot item has a non-empty state difference.

C.10. MatchResultSlot Class

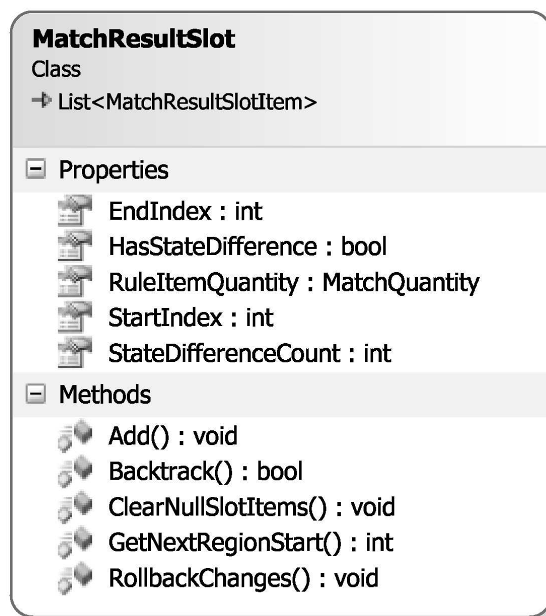


Figure C.10. MatchResultSlot class

This class is a collection of **MatchResultSlotItem** and represents the series of objects matched by a rule item.

The class has the following properties:

- `int StartIndex`: First index of the range of matched tokens in the TokenCollection.
- `int EndIndex`: Last index of the range of matched tokens in the TokenCollection.
- `MatchQuantity RuleItemQuantity`: The quantity specifier of the matched rule item.
- `int StateDifferenceCount`: This is the number of slot items with non-empty state difference.
- `bool HasStateDifference`: This property returns true if the result slot has a non-empty state difference in at least one slot item.

The class has the following methods:

- protected internal new void Add(MatchResultSlotItem item)
  This method adds a new result slot item to the result slot. The parameter *item* is the result slot item to be added to the list.
- protected internal bool Backtrack(ParserState state, out int newIndex)
  This method checks whether a specific rule item can be backtracked and backtracks the slot if possible. The parameter *state* is the current state of the parser object. The parameter *newIndex* returns the new index for applying a new rule item in the input collection, if backtracking is successful. The return value is "true" if the match slot is backtracked successfully.
- protected internal void RollbackChanges(ParserState state, bool rollbackLogItems)
  This method rolls back the state changes for matching a result slot. The parameter *state* is the current state of the parser object. The parameter *rollbackLogItems* is a flag to specify to rollback log items.
- protected internal void ClearNullSlotItems()
  This method removes all slot items with null-reference items.
- protected internal int GetNextRegionStart()
  This method returns starting index of the next region to match by the next rule item.

C.11. MatchResult Class

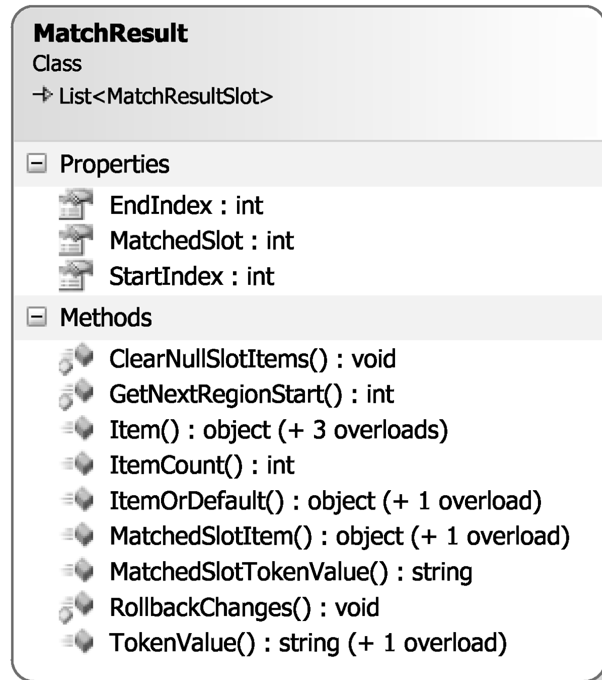


Figure C.11. MatchResult class

This class is a collection of **MatchResultSlot** objects and represents the series of objects matched by a rule. This collection is used to store the matched objects in a grammar rule execution. The matched objects are stored in distinct slots for each rule item. The matched objects can be used in the validator and replace functions.

The class has the following properties:

- `int StartIndex`: This property is the starting index of the matched tokens in the input token collection.
- `int EndIndex`: This property is the end index of the matched tokens in the input token collection.
- `int MatchedSlot`: This property is the index of the matched slot for rules of match type **any**.

The class has the following methods:

- protected internal void RollbackChanges(ParserState state, bool rollbackLog-

Items)

This method rolls back the state changes for matching a rule. The parameter *state* is the current state of the parser object. The parameter *rollbackLogItems* is a flag to specify to rollback log items.

- protected internal void ClearNullSlotItems()

  This method removes all slot items with null-reference items.

- protected internal int GetNextRegionStart()

  This method returns starting index of the next region to match by the next rule.

- public int ItemCount(int slotNumber)

  This method returns the number of matched objects for a rule item. The parameter *slotNumber* is the index of the rule item.

- public object Item(int slotNumber)

  This method returns the single matched object for a rule item. The parameter *slotNumber* is the index of the rule item.

- public T Item<T>(int slotNumber)

  This method returns the single matched strongly typed object for a rule item. The parameter *slotNumber* is the index of the rule item. The type parameter *T* is the type of the object to return.

- public object Item(int slotNumber, int subIndex)

  This method returns the matched object for a rule item. The parameter *slotNumber* is the index of the rule item. The parameter *subIndex* is the index of the matched object in the series of matches for the rule item.

- public T Item<T>(int slotNumber, int subIndex)

  This method returns the matched strongly typed object for a rule item. The parameter *slotNumber* is the index of the rule item. The parameter *subIndex* is the index of the matched object in the series of matches for the rule item. The type parameter *T* is the type of the object to return.

- public object ItemOrDefault(int slotNumber, object defaultValue) This method returns the single matched object for a rule item. If the slot does not contain an item the default value is returned. The parameter *slotNumber* is the index of the rule item. The parameter *defaultValue* is the default value to return if the specified slot does not contain any items.

- public T ItemOrDefault<T>(int slotNumber, T defaultValue) This method returns the single matched strongly typed object for a rule item. If the slot does not contain an item the default value is returned. The parameter *slotNumber* is the index of the rule item. The parameter *defaultValue* is the default value to return if the specified slot does not contain any items.
- public object MatchedSlotItem()

  This method returns the object of the matched slot for rules of match type “any”.
- public T MatchedSlotItem<T>()

  This method returns the strongly typed object of the matched slot for rules of match type “any”.
- public string TokenValue(int slotNumber)

  This method returns the value of the single matched token object for a rule item. The parameter *slotNumber* is the index of the rule item.
- public string TokenValue(int slotNumber, int subIndex)

  This method returns the value of a matched token object for a rule item. The parameter *slotNumber* is the index of the rule item. The parameter *subIndex* is the index of the matched object in the series of matches for the rule item.
- public string MatchedSlotTokenValue()

  This method returns the token value of the matched slot for rules of match type “any”.

APPENDIX D: C# Parser Results

Sample C# source code in file "Arithmetic.cs":

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Text;

namespace CSharpSample
{
    public delegate bool ControlDelegate(object objectToControl);

    public interface ICheckable : IComparable
    {
        bool IsOK();
    }

    public enum EnumMembers {
        Member1,
        Member2,
        Member3,
    }

    public struct Point
    {
        private int x = 0;
        public int X
        {
            get { return x; }
            set { x = value; }
```

```
    }

    private int y = 0;
    public int Y
    {
        get { return y; }
        set { y = value; }
    }
}

public class Arithmetic
{
    private const int width = 5;
    public event EventHandler CustomEvent;

    private int count = 56;
    private string signature = "567";
    protected bool isAvailable;
    static protected internal int internalCounter;
    private static string myStaticName;
    private string myName;

    protected internal bool IsDone()
    {
        return true;
    }

    static public int Add(int a, int b)
    {
        decimal dec = 5m;
        double doub = .5E-27;
        int aa = 5, bb, cc = 6;
```

```
dec += 3;
dec = 0 + 5;
doub = doub + 0;

int sum = a + b;
// add two numbers
a = 0;
a++;

if (a = sum + b)
{
    a = 6;
    dec += 5;
}
else if (a == 6)
    a = 7;
else if (a == 5)
{
    a = 8;
    a = 9;
}
else
{
    a = 9;
    a = 10;
}

a++.ToString();
"hello world".ToString().Length.ToString().Length
    .ToString().ToUpper().ToLowerInvariant();
string s = "";
```

```
    s = s.ToCharArray()[1]--.ToString();
    b = (a * (5 + 4) * 8 / 2) - 1;

    return aa;
    b = 5 + (a=1);
}

[System.Obsolete("Do not use the property MyName.", true)]
[DefaultValue(""), Browsable(false)]
public string MyName
{
    get { return myName; }
    set { myName = value; }
}

static void Main2()
{
    int[] a;
    int[][] aa;
    int[,][,] bbb;

    a[5] = 6;

    char c = ' ';
    decimal dec = 5m;
    System.Console.WriteLine("Hello\n");

    int sum = 0;
    for (int i = 0; i < 50; i=i+1)
    {
        sum += i + 578;
    }
```

```
i = i + 2;

while (sum < 4455)
{
 sum++;
}

try
{
    string helloWorld = "Hello world";
    foreach (string word in helloWorld.Split(' '))
    {
        sum++;
        Console.WriteLine(word);
    }
}
catch (System.IO.FileNotFoundException ex)
{
    sum = 4;
    sum = 5;
}
catch (Exception ex)
{
    sum = 6;
    sum = 7;
}
finally
{
    sum = 8;
    sum = 9;
}
```

```
        }
    }
}
```

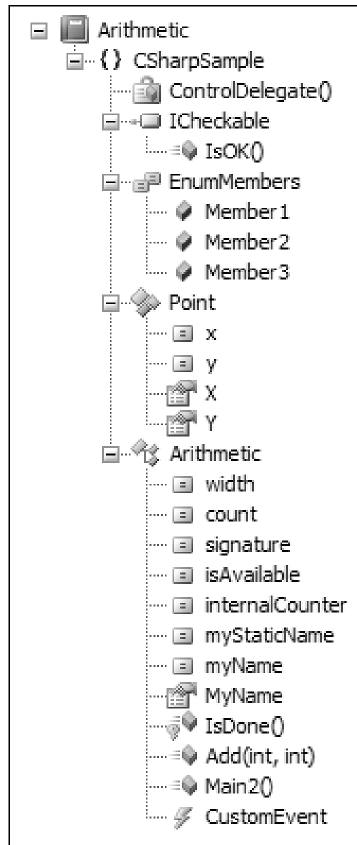


Figure D.1. Model of parsed source Arithmetic.cs

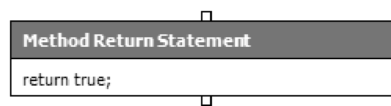


Figure D.2. Flowchart: Arithmetic.cs, IsDone() method of Arithmetic class

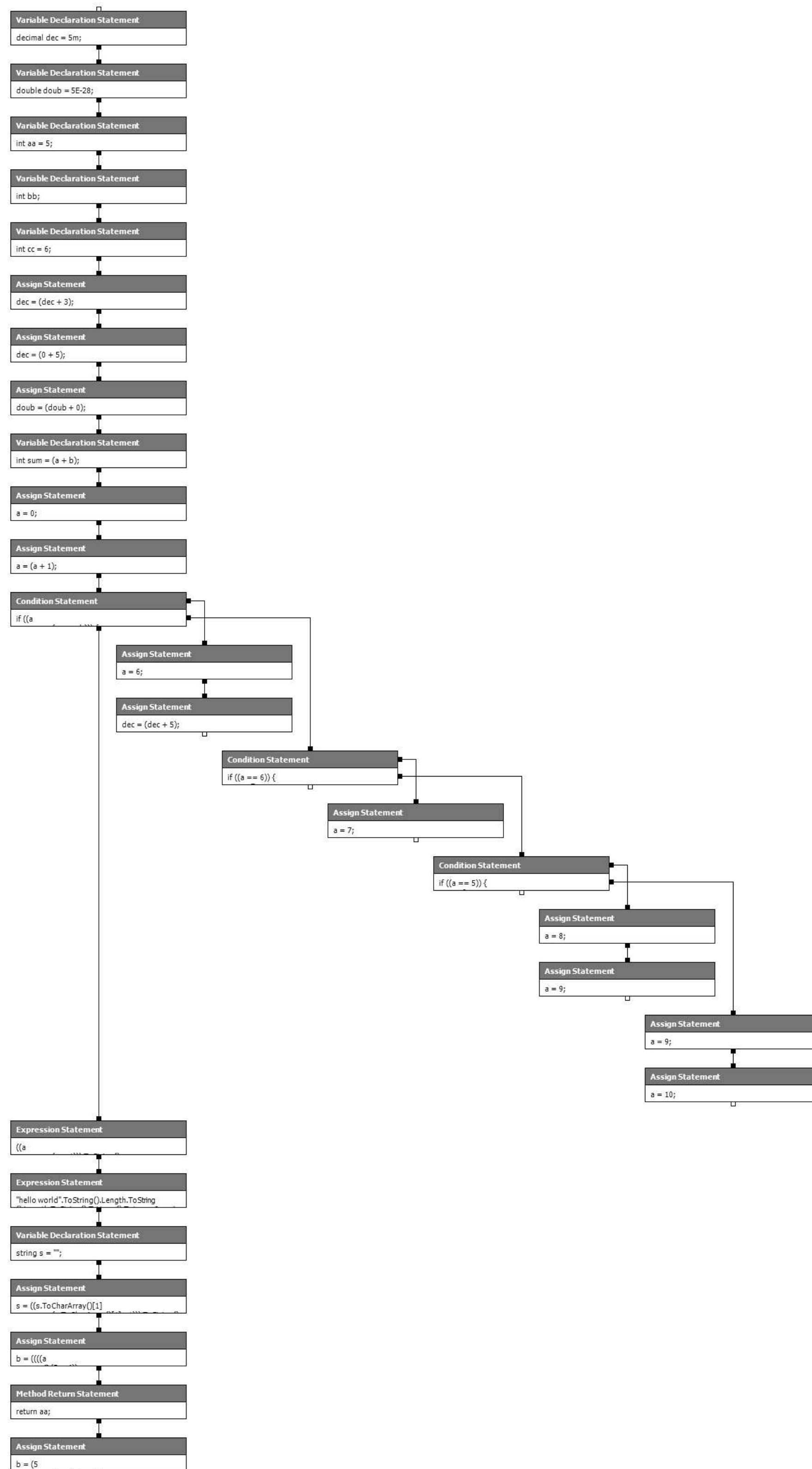


Figure D.3. Flowchart: Arithmetic.cs, Add() method of Arithmetic class

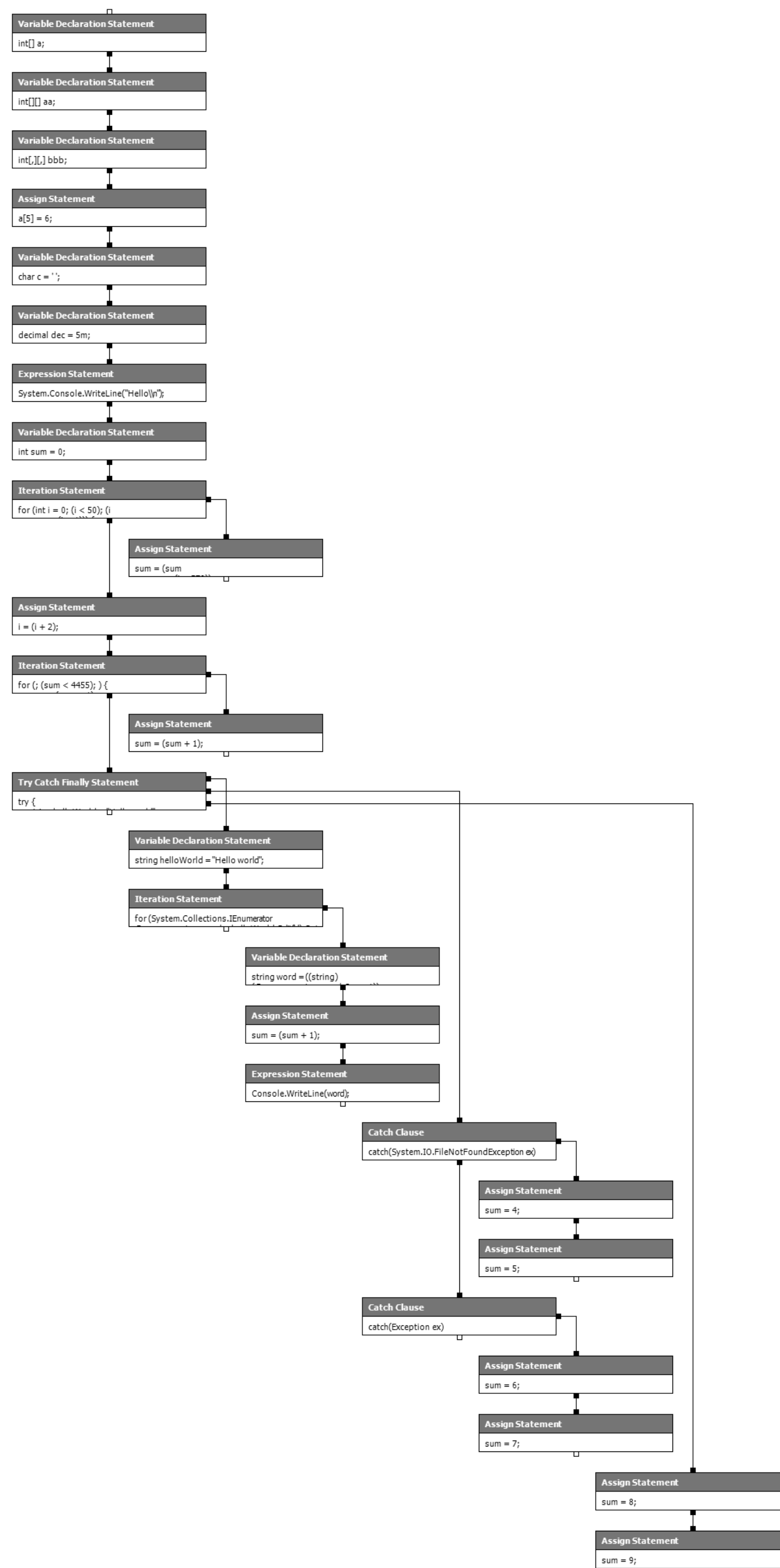


Figure D.4. Flowchart: Arithmetic.cs, Main2() method of Arithmetic class

APPENDIX E: Fortran Parser Results

Sample Fortran 77 source code in file “INTERPOLATION.f”:

```
cccccccccccccccccccccccccc     Program 2.1     cccccccccccccccccccccccc
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                    c
c Please Note:                                                       c
c                                                                    c
c (1) This computer program is part of the book, "An Introduction to c
c     Computational Physics," written by Tao Pang and published and  c
c     copyrighted by Cambridge University Press in 1997.             c
c                                                                    c
c (2) No warranties, express or implied, are made for this program.  c
c                                                                    c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
      PROGRAM INTERPOLATION
C
C Main program for the Lagrange interpolation with the
C Aitken method.
C
      PARAMETER (N=5)
      DIMENSION XI(N),FI(N)
      DATA XI/0.0,0.5,1.0,1.5,2.0/,
     *     FI/1.0,0.938470,0.765198,0.511828,0.223891/
C
      X = 0.9
      CALL AITKEN (N,XI,FI,X,F,DF)
      STOP
```

```
      END
C
      SUBROUTINE TEST_SUB (X)
 550   X = 5.4
      IF (X.GT.7) THEN
        GOTO 550
      ELSEIF (X.GT.6) THEN
        X = X + 1
        X = X * 3
      ELSEIF (X.GT.4) THEN
        X = X + 2
        X = X * 4
      ELSEIF (X.GT.4) THEN
        X = X + 3
        X = X * 5
        IF (X.GT.7) THEN
          GOTO 550
        ELSEIF (X.GT.6) THEN
          DO 100 I = 1, 5
            X = X + 3
  100 CONTINUE
        ENDIF
      ELSE
        X = X + 4
        X = X * 6
      ENDIF
      X = X + 5
      X = X * 7
      END
C
      SUBROUTINE AITKEN (N,XI,FI,X,F,DF)
C
```

```
C Subroutine performing the Lagrange interpolation with the
C Aitken method. F: interpolated value. DF: error estimated.
C
      PARAMETER (NMAX=21)
      DIMENSION XI(N),FI(N),FT(NMAX),IA(NMAX)
C
      IF (N.GT.NMAX) STOP
      DO     100 I = 1, N
        FT(I) = FI(I)
  100 CONTINUE
C
      DO     200 I = 1, N-1
        DO   150 J = 1, N-I
          X1 = 3 +
     *         5
          X1 = XI(J)
          X2 = XI(J+I)
          F1 = FT(J)
          F2 = FT(J+1)
          FT(J) = (X-X1)/(X2-X1)*F2+(X-X2)/(X1-X2)*F1
  150   CONTINUE
  200 CONTINUE
      F = FT(1)
      RETURN
      END
```

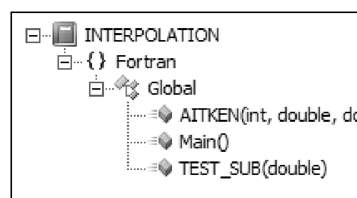


Figure E.1. Model of parsed source INTERPOLATION.f

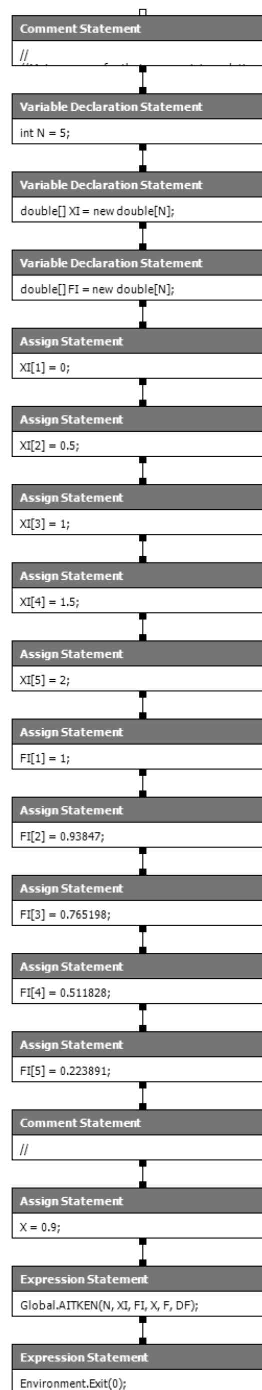


Figure E.2. Flowchart: Main program of INTERPOLATION.f

Figure E.3. Flowchart: Subroutine TEST_SUB of INTERPOLATION.f

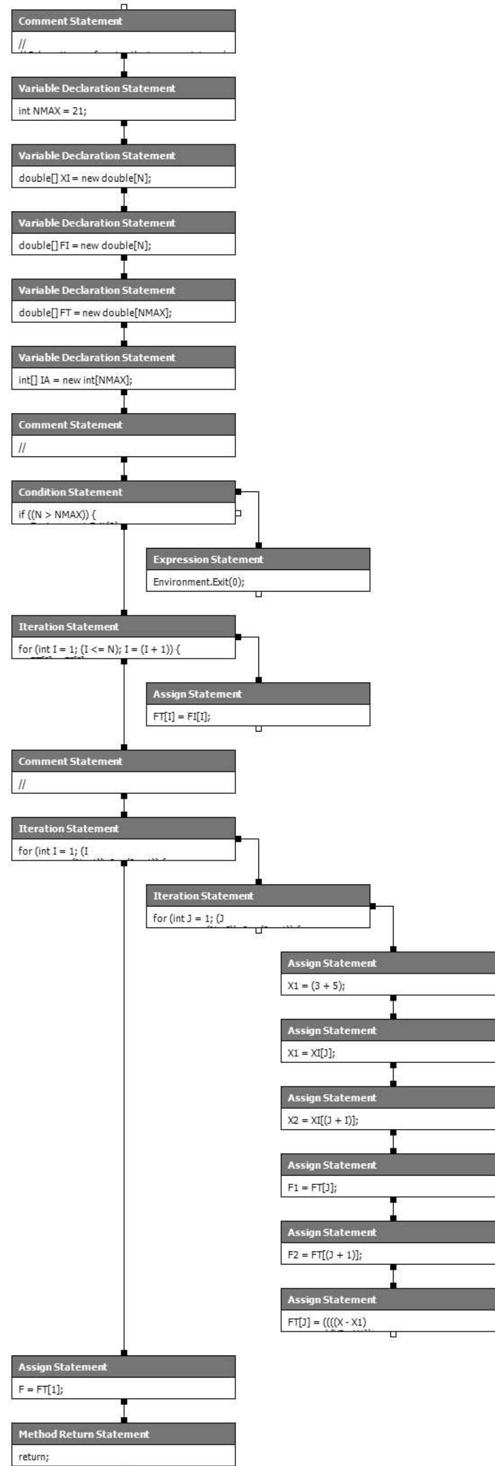


Figure E.4. Flowchart: Subroutine AITKEN of INTERPOLATION.f

REFERENCES

1. Gönen M., "Build.NET – A Graphical Application Generator for Object-Oriented Software and Sample Applications", *Master of Science Thesis*, Boğaziçi University, 2005.

2. *Extensible Markup Language (XML)*, http://www.w3.org/XML/, 2007.

3. *The C# Language*, MSDN Library, http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx, 2007.

4. Lesk M. E., E. Schmidt, "Lex - A Lexical Analyzer Generator", *Computing Science Technical Report No. 39*, Bell Laboratories, Murray Hill, New Jersey, 1975.

5. Johnson S. C., "Yacc: Yet Another Compiler Compiler", *Computing Science Technical Report No. 32*, Bell Laboratories, Murray Hill, New Jersey, 1975.

6. *Yet Another Parser Parser (YAPP) XSLT*, http://www.o-xml.org/yapp/, 2007.

7. *XSL Transformations (XSLT)*, http://www.w3.org/TR/xslt, 2007.

8. Maruyama K., "XML Representations of Source Code to Ease Tool Development", Department of Computer Science, Ritsumeikan University, 2005.

9. *Document Object Model (DOM)*, http://www.w3.org/DOM/, 2007.

10. *XQuery 1.0: An XML Query Language*, http://www.w3.org/TR/xquery/, 2007.

11. *Extensible Software Document Markup Language (XSDML)*, http://www.jtool.org/xsdml.html, 2007.

12. *MoHCA–Java: A Tool for C++ to Java Conversion Support* http://www.cs.ucdavis.edu/~devanbu/dp.tex.pdf, 1999.

13. Devanbu P., "Genoa - a customizable, frontend retargetable source code analysis framework", *ACM Transactions on Software Engineering and Methodology*, (accepted, to appear), 1999.

14. *Visustin v3 Flow chart generator*, http://www.aivosto.com/visustin.html, 2007.

15. *Code Visual 2 Flowchart*, http://www.fatesoft.com/s2f/index.htm, 2007.

16. Badros, G. J., "JavaML: a markup language for Java source code", *Computer Network*, Volume 33, pp. 159–177, 2000.

17. *Overview of SGML Resources*, http://www.w3.org/MarkUp/SGML/, 2007.

18. *System.CodeDom Namespace*, MSDN Library, http://msdn2.microsoft.com/en-us/library/system.codedom.aspx, 2007.

19. *.NET Framework Technologies*, MSDN Library, http://msdn2.microsoft.com/en-us/library/ms644566.aspx, 2007.

20. *.NET Development*, MSDN Library, http://msdn2.microsoft.com/en-us/library/aa139615.aspx, 2007.

21. *Assemblies in the Common Language Runtime*, MSDN Library, http://msdn2.microsoft.com/en-us/library/hk5f40ct.aspx, 2007.

22. *System Namespace*, MSDN Library, http://msdn2.microsoft.com/en-us/library/System.aspx, 2007.

23. *System.Xml Namespace*, MSDN Library, http://msdn2.microsoft.com/en-us/library/System.Xml.aspx, 2007.

24. *.NET Framework Regular Expressions*, MSDN Library, http://msdn2.microsoft.com/en-us/library/hs600312.aspx, 2007.

25. *XmlSerializer Class*, MSDN Library,
http://msdn2.microsoft.com/en-us/library/System.Xml.Serialization.
XmlSerializer.aspx, 2007.

26. *XML Schema*, http://www.w3.org/XML/Schema, 2007.

27. *CodeCompileUnit Class*, MSDN Library,
http://msdn2.microsoft.com/en-us/library/system.codedom.
codecompileunit.aspx, 2007.

28. *CompilerParameters.GenerateInMemory Property*, MSDN Library,
http://msdn2.microsoft.com/en-us/library/system.codedom.compiler.
compilerparameters.generateinmemory.aspx, 2007.

REFERENCES NOT CITED

1. *A Compact Guide to Lex & Yacc*, http://www.epaperpress.com/lexandyacc/download/lexyacc.pdf, 2007.

2. *Java–XML Tools Project*, http://www.jtool.org/, 2007.

3. Rayside D., M. Litoiu, M. A. Storey, C. Best, R. Lintern, "Visualizing Flow Diagrams in WebSphere Studio Using SHriMP Views", *Information Systems Frontiers*, Volume 5, Issue 2 (April 2003), pp. 161–174, 2003.

4. Purtilo J. M., E. L. White, "A flexible program adaptation system: Case studies in Ada", *Journal of Systems and Software*, Volume 17, Issue 2 (February 1992), pp. 129–143, 1992.