FLEXIBLE CODE CHECKER

by

M. Erdal ŞEKERCİ

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

FLEXIBLE CODE CHECKER

APPROVED BY:

Prof. H. Levent Akın (Thesis Supervisor)	.....
Assoc. Prof. Can Özturan	.....
Assoc. Prof. Ali Tamer Ünal	.....

DATE OF APPROVAL: 15.06.2007

ACKNOWLEDGEMENTS

First of all, I am grateful to my thesis supervisor Prof. H. Levent Akın for privileging me to work with him. His guidance and support throughout this study made things easier for me.

I would like to express my gratitude to Assoc. Prof. Can Özturan and Assoc. Prof. Ali Tamer Ünal for taking part in my thesis jury.

I would like to thank my colleague Selim for all of the delightful times we have spent together.

Finally, I want to thank my family for their continuous support and encouragement. This could not be true without them.

ABSTRACT

FLEXIBLE CODE CHECKER

In software system development, the most important issue is to build a system that satisfies the requirements and works correctly. But even if a software system works correctly, this does not show that its source code is well-written. The source code may contain unnecessary codes, may have undetected bugs, or may not conform to the required coding standards or rules. This will make the code dirty and unreadable making it very difficult for other people (except the writer) to understand, update or analyze the source code. The motivation for this thesis underlies in this issue. The idea is to analyze a given source code and check it according to user defined flexible checker rules and coding standards (conventions). This could be useful in checking source codes in a variety of areas such as student projects in programming courses and deciding how good the source codes were written according to the defined rules of the checker.

ÖZET

ESNEK KOD DENETİMİ

Yazılım geliştirme sürecinde üzerinde durulan en önemli nokta, gereksinimleri karşılayan ve doğru şekilde çalışan bir sistem üretmektir. Ancak, bir yazılımın doğru şekilde çalışması, onun kaynak kodunun kaliteli olduğunu ve düzgün yazıldığını göstermez. Kaynak kod içersinde fark edilmemiş hatalar veya gereksiz kodlar bulunabilir yada kaynak kod belirlenmiş kodlama standart ve kurallarına uymayabilir. Bu tip durumlar kaynak kodunun okunaksız bir hale gelmesine yol açar ve kaynak kodunu yazan kişi dışındaki kişiler tarafından anlaşılmasını, geliştirilmesini ve analiz edilmesini oldukça zor hale getirir. Bu tez çalışmasına kaynaklık eden en önemli nokta da bu tip durumları engellemektir. Bu çalışmadaki fikir, verilen bir kaynak kodunu analiz etmek ve onu kullanıcı tarafından tanımlanan esnek kontrol kurallarına ve kaynak kod standartlarına göre denetlemektir. Böylece, programlama derslerindeki projeler gibi çeşitli alanlarda verilen bir kaynak kodunu denetlemek ve kaynak kodunun tanımlanan kurallara göre kalitesini belirlemek mümkün olacaktır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BNF	Backus-Naur Form
C#	C# programming language
CDML	Checker Rule Definition Markup Language
CodeDOM	Code Document Object Model
DOM	Document Object Model
GDML	Grammar Definition Markup Language
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
MSDN	Microsoft Developer Network
XML	Extensible Markup Language

1. INTRODUCTION

The basic steps in software development cycle are requirements elicitation, design, implementation and testing. The implementation phase includes the conversion of formal designs and specifications into source code by utilizing a programming language. The most important concern in this phase is to build a reliable system that works correctly and satisfies the requirements. Thus, programmers generally focus on these issues. On the other hand, the quality of the source code is another important issue in the programming phase. Code quality is crucial in source code maintenance and analysis. However, programmers usually pay little attention to it.

Code quality is important in many aspects. Analyzing a well-written source code is a far easier task than analyzing a code that is full of unnecessary or untidy code segments, even if both do the same task. Thus, the source code of a correctly working program may not be well-written. Furthermore, the maintenance of a bad-written code is really difficult. A programmer that needs to update the code or continue the development on the software must first understand the code which can be a daunting task for any programmer. In fact, only the author of the code can understand this code easily. As a result, the maintenance of the system can really be hard and costly. Therefore, a good software system needs to have a well-written source code besides reliable and correct operation. For this purpose, source code checker systems have been proposed. A source code checker analyzes a given source code and identifies possible violations against the rules of the checker. There exists a wide range of rules for checking a source code, including naming conventions, indentation controls, variable initialization and usage checks, memory allocation checks, security flaws etc.

In this thesis, a generic source code checker framework is proposed. The flexible code checker is designed to analyze a given source code against the user defined rules of the checker. The motivation for this thesis is to analyze the source codes of students in programming courses and grade the source codes according to the quality of the source code. The rules for the analysis of the code are flexible and are defined by the

user via the graphical interface of the code checker. The user defines the rules of the checker as methods by defining the conditions to be satisfied by the code fragments and the checks to be done on the code items. Each rule of the checker has an associated importance defined with it, denoting its importance level in the code analysis. The checker analyzes the source code according to these checker rules and identifies the violations of these rules for the given source the code. The violations are displayed based on the importance of the violated rules, showing the quality of the source code according to the checker rules.

In order to build this code checker, a generic parser framework that is used for generating a parser from a given grammar of the target language is proposed. This parser framework generates a parser for the target language that contains the grammar information of the target language and a checker that contains the checker rule information to make source code checking with the generated parser with the associated checker rules in the generated checker. The required inputs for this generic parser are the grammar of the target language, which defines the grammar items and the methods for parsing the target language code, and the checker rules for the checker on the grammar of the target language. Hence, the rules of the checker are defined by the user providing a flexible framework.

In the next chapter, the topic of code analysis and the related work on code checking in the literature is summarized. There exist also commercial tools for source code checking. Some of them are also shortly reviewed. Then in chapter 3, design and implementation details of the proposed flexible code checker are given. Chapter 4 summarizes the sample applications of the checker framework on two target programming languages, C# and C++, for building code checkers for these languages. Finally, conclusions and future research directions are discussed in chapter 5.

2. CODE ANALYSIS

Software maintenance is one of the most expensive and time consuming phases in the software life-cycle. However, despite its importance, it is given little emphasis by software developers. In reality, maintenance of the software is not quite simple as it is perceived by software developers. The cost of software maintenance is estimated to be around 70 percent of the total life-cycle cost [1]. As a result, the need for code analysis during the maintenance and evolution of existing software systems is widely recognized by maintenance programmers and designers.

Even if a software system compiles and works correctly, it does not show that its source code is well-written and is maintainable. Software systems may have some defects that reduce the maintainability and that may not be recognized by the compilers. In order to identify these defects, some form of code analysis is necessary.

The code analysis approaches basically can be grouped in two categories: dynamic analysis and static analysis. Dynamic analysis comprises the analysis methods that execute the program while making the analysis. In dynamic analysis, actual behavior is compared with the expected behavior for testing the software. However dynamic analysis can be difficult and costly depending on the complexity of the software.

On the other hand, static analysis approaches do not execute the program for analysis; they go over the source of the program trying to match certain patterns for identifying potential bugs.

The rest of this chapter is organized as follows: Static source code analysis is discussed in section 2.1. Then, in section 2.2, related work on the topic of code checking is covered.

2.1. Static Code Analysis

Static Code Analysis is a process of checking programs for errors without executing them. It covers the area of performing all kinds of tests on the source code of programs. Static analysis is performed to improve and ensure the quality of source codes of software programs.

The tools utilizing static code analysis are called static checkers. They examine the source code of the programs looking for certain known error patterns and coding standard violations. Hence, the power of static checkers is limited by the definition of error patterns or coding standard violations of the checker. Thus, the static checkers do not verify the correctness of programs; they help in identifying certain bugs and violations. On the other hand, the earlier the bugs are identified; it is easier to fix them. Hence, catching certain coding errors or violations during implementation and before the testing phase would be better.

The static checkers are usually applied after compilation and before testing. Hence, a static checker assumes the source is precompiled and makes the checking on a syntactically correct source code. They apply the defined error patterns and standards on the source for identifying bugs and violations. A static checker basically takes a program source and constructs an abstract model representing it. Then it uses this model for checking the code according to the rules of the checker and identifies the violations.

In traditional software design, the basic software design lifecycle consists of code editing (implementation), compilation, linking and testing. The traditional software design cycle does not include code analysis as a phase of the lifecycle; it is the responsibility of the programmer to write readable and maintainable code. The traditional software design lifecycle is depicted in figure 2.1. The programmer develops and compiles the source code and tests the resulting program until a desirable and working program is obtained.

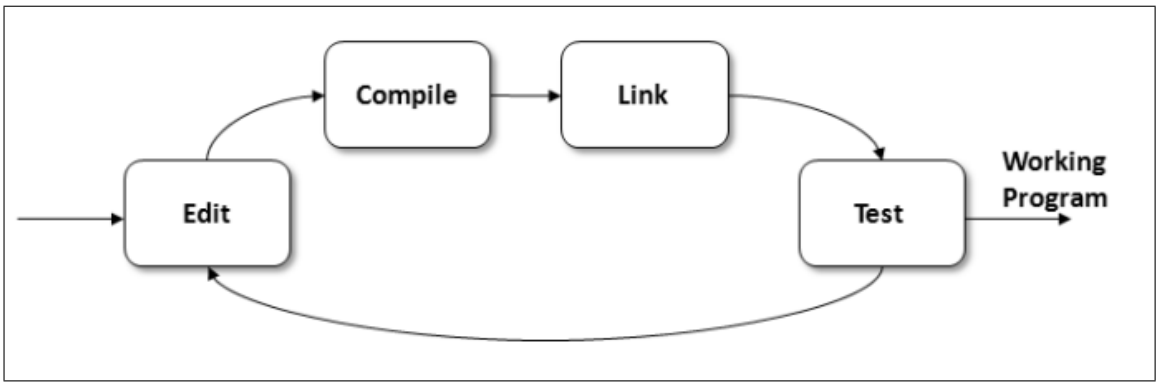

Figure 2.1. Traditional software design lifecycle

With the recognition of the need for code analysis in the evolution of software systems by the developers, code analysis is introduced into the software design lifecycle. The visionary software design lifecycle includes code checking after the compilation phase of the lifecycle. It is depicted in figure 2.2. The programmer develops, compiles and then analyzes the source code before the testing phase to obtain a clean code in terms of the analysis phase. Then the resulting code is linked and the resulting program is tested until obtaining the desired working program.

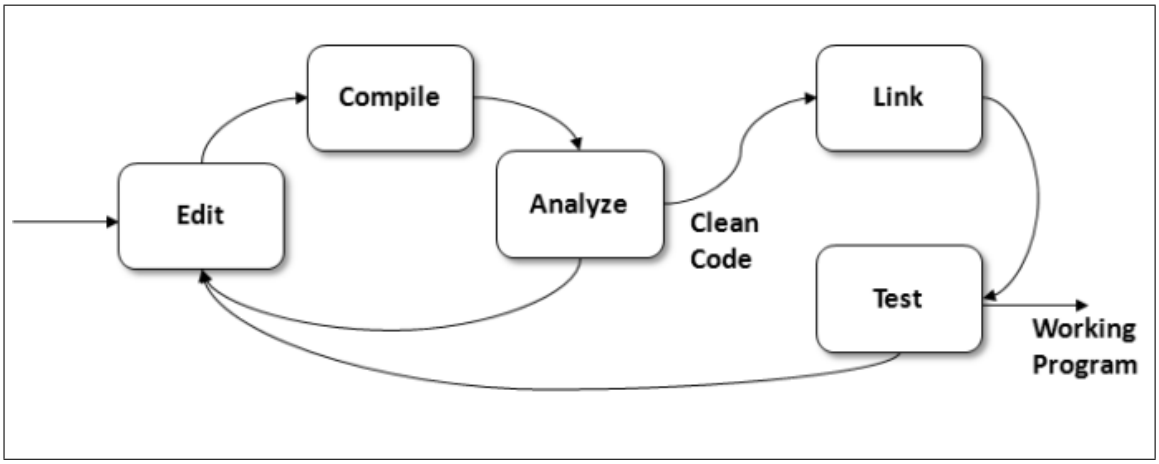

Figure 2.2. Visionary software design lifecycle

A code checker helps in enforcing coding standards by detecting violations against the standards. A coding standard defines the way a programmer must and should write the source code of the program. The rules of the checker describe a coding standard for the target language. The result of the analysis shows how much the program adheres to the coding standard, a measure of the quality of the source code. Coding standards are useful in improving the readability, reliability and maintainability of source codes.

Although adoption of coding standards is difficult, coding standards are necessary for improving the code quality. On the other hand, no code checker can assure the correctness of a program, it helps in detecting certain errors in the program that are not detected by the compiler.

2.2. Related Work

There have been a number of proposals devised related to the problem of code checking in the literature, some of them in research or conference papers and some as a part of commercial tools provided. Most of these have been proposed for specific purposes and cannot be used for general purpose checking. They are briefly summarized in this section.

The problem of checking and grading student software codes considering their style has not been a wide research topic. Ala-Mutka, Uimonen and Jarvinen [2], have discussed the topic thoroughly and have proposed an approach for automatic checking of programming projects for specific coding styles. They focused on programming style and argued that it is a quality measure for software and it strongly affects readability and maintainability of the code (especially for bigger projects). They discussed that students perceive programming style as a secondary issue and generally pay little or none at all attention to it. Furthermore, they argue that students should not be allowed to use bad convention habits freely, because it can be difficult to change that later. But checking coding style in programming courses is not easy since it takes a lot of time and generally the size of the class is large. Thus, a software tool for this purpose is required.

For this purpose, they have proposed and implemented a tool, called Style++, for analyzing C++ programming style which can be used by students and instructors at their university for C++ code assessment. They have collected a set of coding rules and conventions for this purpose and implemented the analyzer based on these style rules. In the implementation, they have omitted the features that can be analyzed by compilers with full warning options. The features that are checked can be categorized

as follows: modularity (various C++ class implementation issues, general encapsulation issues), typography (commenting, naming conventions and code layout), clarity (average length of functions, blocks and short-circuit statements), independence (correct return values and avoiding numeric literals), effectiveness (small variable scopes) and reliability (float comparisons, default case for switch statements).

For the assessment of codes, the instructors have the ability of deciding the contribution of each feature (assigning a weight for each feature), thus the tool is customizable for different practices. In their analysis of related work, they have stated that it is difficult to find assessment approaches for the C++ language and they could not find any available assessment tools for C++ for educational purposes in the literature. The implementation has been used in the university and the results have shown that the tool is useful and the students have started to pay more attention for the coding style. Thus the quality of students' coursework has been improved after the adoption of Style++.

Another approach on grading student programs has been discussed by Jackson and Usher [3]. They have built a software tool with a graphical user interface, called ASSYST (Assessment System), for grading student programs. They have argued in their discussion that grading student programs is a laborious and long process and which is subject to human errors. They have identified that some areas that automatic checking of codes is possible. For this purpose, they have built a tool in which the assessment process (grading) is directed by the user. The tool attempts to check the correctness (by checking the outputs of a program against a given output pattern), efficiency (running time), complexity and style. The tool was applied to a few courses and the initial reactions were positive.

Holzmann [4], have presented a nice analysis of source code checking tools and have proposed a static source code checking tool called UNO. He argued that developing a tool that can detect all defects in a program with certainty is impossible and instead of trying to build a tool that tries to detect all types of errors, it is better to concentrate on more common errors for a specific area. This would lead to better signal to noise ratio

and hence better performance. The UNO tool checks three types of software faults in C codes by default (where its name comes from): Uninitialized variables, dereferencing Nil-pointers and Out-of-bound array index checking. The other faults to check are completely user-controlled, that is the user can define application specific controls to check for via simple functions. Thus, the tool is extendible with user-defined properties which should be written in C, the language of the tool, and this feature increases the power of the tool but at the same time requires user assistance for code checking. The UNO tool uses an external parser, namely the parse tree output by the ctree compiler, and makes the analysis using the parse tree.

In his work, Holzmann [4] also included an analysis of commercial code analyzers available. He presented that checks done by a compiler cannot be enough for a detailed checking of source code, thus special purpose analyzers are necessary. An early example of a code check tool is lint, which dates back to 1977, but provides limited checking. A recent extension for it is lclint, which provides significant improvements. Another tool in this area is Microsoft's PREfix that targets large code bases. He also argued that the checking power of code checker can significantly be increased if more information on the code is provided by the programmer. An example for this type of checker is Microsoft's Vault system for C code.

Clarke, Kroening and Lerda [5], have proposed a tool for formal verification of C programs that uses Bounded Model Checking (BMC). They argue that formal verification of C code is challenging due to the usage of arithmetic, pointers, bitwise operators and pointer arithmetic; and that security of these applications is critical for low level applications. They have described a tool that formally verifies C programs in their work. The tool checks for safe pointer usage, array bounds, dynamic memory allocation, recursion and float and double data types. The tool utilizes a technique called Bounded Model Checking that converts the C program into a Boolean formula which is satisfiable if and only if there is an error in the program. Afterwards the formula is checked for satisfiability. The results are then shown and can be traced in a graphical user interface.

Potrich and Tonella [6] have presented an open architecture for the analysis of C/C++ code. It discusses the model for the analysis of a source code, and gives details on the architecture. The architecture discusses a different view for the analyzer and a complier, and explains that the analyzer extracts higher level information from the code and does not require a performance comparable with that of a compiler since the analysis of C/C++ code is really a complex task. The model contains a parser for parsing and a separate package for the rules allowing for addition of new rules. The first objective was development of a coding rule checking tool, while the future work is a reverse engineering module.

O'Callahan and Jackson [7] have developed a program understanding tool, called Lackwit, and is a tool based on type inference for analyzing a C program efficiently. They use type inference since they claim that it is automatic and can handle complex language features of C such as recursive pointer based structures and pointers of functions. The designed tool, Lackwit, is shown to be very efficient. They claim that Lackwit has advantages over existing tools for program understanding. It has been compared with another code checking tool, namely LCLint and is found to be complementary with it for code checking (Each has advantages over the other).

The problem of unused components in source code is discussed in [8]. In this research, Di Penta, Neteler, Antoniol and Merlo have proposed a framework and a tool that provides removal of unused and unnecessary codes. They present that several factors such as unused code components, code clones which were added for maintenance purposes, non-systematic addition of new functionalities that increase the software complexity and reduce the software quality. Furthermore, these factors increase the application sizes and memory requirements significantly and systems become difficult to maintain. In order to control some of the above quality factors, they proposed a framework called as the Software Renovation Framework (SRF). It performs refactoring on the code, removing unused and unnecessary code segments, monitoring or factoring out code clones and restructuring of libraries and objects. In this way the software size and complexity will be reduced and the code will become more manageable. The framework analyzes the dependencies among executables and object files which can

be represented as a dependency graph. Thus, the software miniaturization (minimization) problem can be modeled as a graph-partitioning problem which is known to be NP-hard. Thus heuristic techniques were used for obtaining a good enough solution. The proposed approach uses Genetic Algorithms as a heuristic approach but also it is stated that a hybrid approach of genetic algorithms and hill climbing would give better performance. Furthermore, SRF is applied on C/C++ but it is proposed as a language independent framework. The result of the analysis stated that SRF performs quite well and helps to monitor and improve the quality of a software system.

In [9], Reig proposes a flow-sensitive type checking algorithm for checking certain programming errors in C programs that may cause security vulnerabilities. They present that automatic detection of such errors are necessary because manual code inspection would not be suitable for large projects and many errors could be missed in these checks. Some of these errors include dereferencing an invalid pointer, double-free of pointers, reading uninitialized memory, and invalid deallocation of memory. These errors may cause security vulnerabilities such as program crashes or denial of service attacks. They have proposed a novel type checking algorithm for the detection of many of these rules. Furthermore the proposed approach allows the use of temporal rules, enabling/disabling of rules. The proposed algorithm checks for the types of expressions against the constraints, and the constraints are propagated transitively and inter-procedurally.

Deeprasertkul, Bhattarakosol and O'Brien [10], discuss the importance of software reliability and propose a technique called Precompiled Fault Detection (PFD) for increasing the software reliability without increasing the programming responsibilities. They argue that software reliability is very critical and some programming errors of software developers cannot be detected by compilers which in turn can cause runtime failures, causing unreliability. Developers try to prevent those programming errors and achieve reliable software during development, but still some errors cannot be detected. Hence they argue that automatic detection of these faults is a challenge for researchers and many techniques have been proposed. They propose the precompiled fault detection (PFD) technique for checking faults in source code before compilation. The

technique was applied for C programs. They claim that hidden faults (that are not detected by the compiler) occur rarely but an occurrence will cause critical system failures. They have used the following hidden faults in C programs as a model for detection of faults: out of bound array indexes, mismatches between actual and formal parameters in function calls, nonterminating loops and use of uninitialized variables. PFD was implemented using the C language and consists of two parts: detection and correction. For validation of the technique they have used the following faults: out of bound array indexes, mismatches in function arguments and no-default case in switch statements. They have conducted experiments based on these faults and shown that the PFD technique significantly reduces the number of faults and increases the software reliability. They claim that the PFD technique is not limited to the above faults only, it has wide applicability and that they will apply this technique to other programming languages.

Another problem discussed in research papers is extracting syntactic information from source code during the development process. C.Depradine [11] discusses the problem and proposes CITOR, Code Information Extractor, an expert system for extracting syntactical information from Java code. They argue that extraction of syntactic information (program structure information) from source code cannot be done until the code is near completion for many tools, which in fact can be too late. The syntactic information can be used for various tasks such as syntax checking and code convention maintenance. We know that conventions are very important for readability and software maintenance. For this purpose, they proposed CITOR; a rule based expert system for extracting information from Java code. It provides code and design maintenance; and discovery of class names, field declarations, variable declarations, constructors and exceptions. The experimental results have shown that using an expert system can be useful in this context but the results should be used together with other information for better controls in real time.

Van de Vanter has provided an analysis of documentary structure of source code in his work [12]. They argue that design of language tools should take the documentary structure into account and discuss that analyzing the documentary structure of source

code provides better analysis of requirements of programming tools. A documentary structure includes the information regarding the readability of the code such as comments, white space and names; in addition to the language text. They examine the documentary structure and its consequences for the design of language based tools. The documentary structure consists of textual aspects of the code that are not part of the programming language: comments, white space and choice for the names. The documentary structure is added to the source code for aiding the human reader and thus very important for the readability of the code. They treat the programmers as authors taking responsible for the readability of the source code.

In [13], Jorgensen provides an analysis on the measurement of software quality. He presents that no universal quality measure exists for measuring the software quality, but there are some meaningful measures for the indicators that are the indicators of software quality. Software quality can be measured in terms of a set of quality factors or user satisfaction or by the unexpected behaviors (or errors) of the software. Each of these measures the software quality from a different perspective of user needs and may lead to different results. As a result of the analysis, he recommends that instead of trying to develop a widely accepted quality measure, one should focus on better measures for quality related properties such as user satisfaction and error density. Furthermore, the results should not be interpreted as a direct quality measures since it can be misleading.

In [14], Pollet and Le Charlier propose and design a Java code analyzer based on an abstract interpretation technique. They attempt to provide a generic framework for analysis. They use a representative subset of the Java language for analysis, they exclude concurrency features. The proposed framework uses structural abstract domains and is seen as a first step for an abstract interpretation framework for Java. This work can be extended in a variety of ways such as investigating variants of the abstract domains and extending this work to complete Java language which are indeed given as future work.

LCLint, which is an extension of the lint, is a famous available tool for code

checking. It was proposed in [15] by Evans, Guttag, Horning and Meng Tan. They have proposed LCLint, an efficient and flexible tool for checking C programs. However, it requires user involvement, since it requires a specification for better analysis. Without a specification it cannot perform better than traditional lint. The specifications should be written in the LCL language. LCLint is a command line tool that reports inconsistencies between the code and a combination of specification and the programming conventions. The design goals considered in the design of LCLint are efficiency (checking time), flexibility (customization of checking), incremental effort (more specification yields better check) and ease of use. Flexibility of the tool is provided via various command line flags.

Splint (Secure Programming lint) [16] is a successor of LCLint, and developed and maintained by the Secure Programming Group at the University of Virginia Department of Computer Science. The splint tool is a freely available tool. It is useful for static checking C programs for security vulnerabilities and programming mistakes. Problems detected by Splint include: dereferencing a possibly null pointer, using possibly undefined storage, type mismatches, violations of information hiding, memory management errors including uses of dangling references and memory leaks, dangerous aliasing, problematic control flow such as likely infinite loops, fall through cases or incomplete switches, buffer overflow vulnerabilities and violations of customized naming conventions.

Another commercial tool for checking C code is Microsoft's PREfast tool [17]. PREfast (successor of PREfix) is a static analysis tool for C/C++ that detects certain kinds of errors in source code, which are not easily found by a typical compiler. It simulates execution of all possible code paths on a function-by-function basis and checks each possible path against a set of rules that identify potential errors or bad coding practices, and then gives warnings for the parts breaking the rules. The code needs to be compiled in order to be analyzed by the PREfast tool. It can detect several significant categories of errors including memory, resources, function usage, coding practices and precedence rules. Another available tool for C++ code checking is Cxxchecker [18] which checks C++ code for common coding styles. Cxxchecker, a command line tool,

has been implemented in Python and uses gccxml for parsing the code. The coding styles that are checked include variable/class/function naming consistencies.

3. FLEXIBLE CODE CHECKER

In this study, a general purpose and flexible source code checker framework is designed and implemented. The designed source code checker is capable of analyzing a given source code according to the given grammar of the target language and user defined rules of the checker. The target grammar and rules of the checker are input to the checker and are defined by the user. As a result of the analysis, the violated checker rules are presented to the user based on the user-defined importance of the rules, which in turn can be used for grading the code. Thus, the code checker can also be applied to source code grading, which is indeed the motivation for this thesis.

3.1. Requirements

Commercial code checkers mostly are used for achieving code standardization which is an important factor of code quality. Coding standards are useful for obtaining consistent, more readable and maintainable codes. They are not necessary for the success of software development but they can make the analysis and the maintenance of source code easier.

Therefore, the first requirement for a general purpose code checker is the flexibility and the adaptability of the checker for different coding standards. The checker should be flexible in managing the rules of the checker for the analysis. The user should be able to build his/her own schema for the rules of the code checker. Thus, a flexible checker is necessary in which the user is able to build custom rules for the checker.

In order to carry out the analysis, the code checker first has to parse the source code for identifying the building blocks in the source code such as statements, blocks, functions, variables and expressions etc. Therefore, the code checker requires a good and efficient parser for the target programming languages. The parser requires the grammar of the target language in a convenient form in order to make the parsing and analysis easier. In order to provide more flexibility, the checker should be made

independent from the target languages as much as possible. This will make the addition of new programming languages for the checker much easier.

Furthermore, the results of the analysis should be presented to the user in a convenient way. Each violation of the rules of the checker should be listed in a convenient form based on the importance of the rules and the user should be able to see where the violations occur. These results can be used for grading the code, thus measuring the quality of the code, according to the importance of these violated rules. The importance of the rules should also be managed by the user.

3.2. Target Users and Application Areas

The designed code checker system targets software developers for helping them in analyzing their source codes and find the errors that they would like to check. For this purpose, the user defines the grammar for the target language that specifies how to parse a given source code in that language. Therefore, adding a new language to the checker is done by defining its grammar. Hence, a grammar definition corresponds to a new language in the checker.

For every target language, the user defines the set of rules to be checked in the source code. The user is able to define different sets of checker rules for a single language enabling the user to check different set of coding standards on the same language. In order to analyze a source code, the user has to select the grammar definition of the target language and the checker rule base for the analysis. Then the checker analyzes the given source code based on the grammar and checker rules and presents the violations of the checker rules.

The proposed system requires the grammar definition and checker rule definitions from the user giving the freedom to the user for building totally customized code checkers. The user can build different rule bases (coding standards) for a single language grammar and use them in different contexts. Therefore, the checker is completely user-defined and it provides the flexibility to the checker.

The complexity of the code checker depends on the user defined grammar and code checking rules. Checking for a complex error pattern or a convention in the source code may require a complex rule definition in the checker, which increases the complexity of the checker. Furthermore, building the checker rule for such an item may not be worth checking and may become infeasible.

The code checker system can be used by all programmers for analyzing their codes by their custom rules and finding out the violations of these rules. The results of the check, the list of violations of the checker rules based on their user-defined importance, present the quality of the source code according to the rules of the checker. The checker can also be used by instructors for grading student programs. Using the results of the check, a code quality grade can be calculated. Furthermore, the checker can be used in software projects for increasing code quality and maintainability of the source codes in the project.

3.3. Design

In this thesis, a general purpose flexible source code checking framework is designed and implemented. Using the framework, the users can build source code checkers for programming languages by defining the grammar for a target language and checker rule bases for checking a given source code for the target language. The framework is designed as a flexible one in order to meet the requirements given in section 3.1.

The designed source code checker is made up of three stages basically: *Tokenizer*, *Parser* and *Analyzer*. These stages are depicted in figure 3.1. The *Analyzer* stage is actually a part of the *Parser* stage, due to the fact that the analysis is done during the parse operation. The inputs for the Source Code Checker are the Grammar definition of the target language and the Checker Rules definition for the analysis of the source code checker specifying the checker rules.

The designed code checker takes a given source code and first parses it for obtaining language specific tokens according to the tokens defined in the grammar of

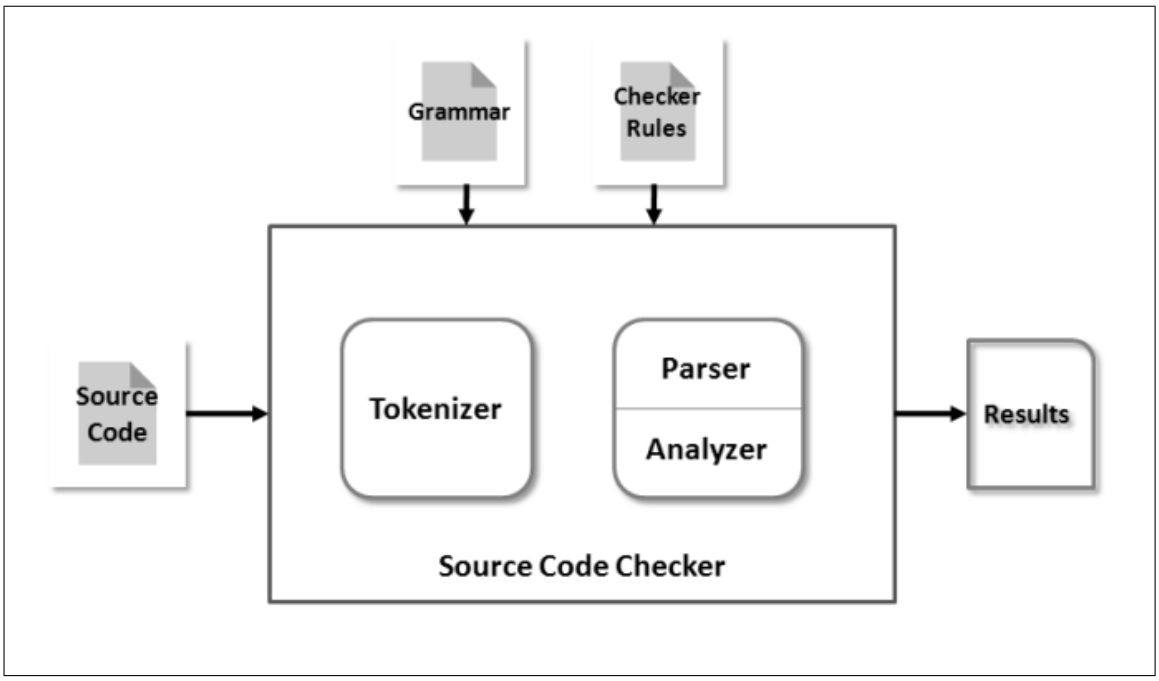

Figure 3.1. Stages of the code checker

the target language. The grammar of the language is supplied in a special XML base form to the code checker. This form is called as the **Grammar Definition Markup Language**, referred as *GDML*, and is defined in section 3.4 in detail. The grammar basically includes the tokens and syntactic rules of the language along with the transformation methods to use in the parsing stage for generating the parse tree model of a given source code to be used in the analysis.

On the other hand, the checker rules are defined in another special XML base format, called as the **Checker Rule Definition Markup Language** (*CDML*). This format is similar to the GDML format and is discussed in section 3.5 in detail. The checker rules consist of the analysis method definitions on the rules of a target GDML grammar that specifies the conditions to check. The analysis methods are completely defined by the user and provide the flexibility to the user to define custom rules according to the needs.

Hence by defining the target grammar and the checker rules, the user creates a custom checker for the target programming language. The user is also able to define different checker rules for a target grammar of a language, providing the checking with

different coding rules and standards on a language.

Thus, the initial requirement of the checker for analyzing a given source code is to prepare the parser for the target language using the language's grammar. The generic parser of the checker reads the given grammar of the target language and prepares the language specific parser class that includes the necessary parsing rules for the language. Hence, the language specific parser is automatically generated by the checker using the CodeDOM libraries of Microsoft .NET Framework [19]. Code Document Object Model (*CodeDOM*) is shortly reviewed in section 3.3.1. The generated parser is used for the tokenization and parsing of a given source code.

In the *Tokenizer* stage of the checker, the source code is converted to a list of language specific tokens defined in the given grammar. A token is the simplest language block in the source code such as keyword, identifier, operator, comment, whitespace, literal etc.

In the next stage the tokenized code is parsed for obtaining the high level parse tree of the source code. For instance, in the *Parser* stage, the tokens are combined to obtain high level code phrases such as expressions, statements, functions, loop blocks etc. The parse tree contains the tokens at the leaf nodes and the model definition of the language defined in the entry rule of the grammar at the root node.

The parser constructs the parse tree using the rules specified in the grammar starting with the entry rule and the methods defined in the grammar for transforming lower level objects to higher level ones. For instance, the rules define the list of items at level *(n+1)* of the tree, to match as children of a node at level *(n)*; whereas the transformation methods in the rules define the transformation of the child items at level *(n+1)* to the parent node at level *(n)*. Hence, the level of code phrases increases from leaf nodes to the root node in the parse tree (i.e. *tokens-expressions- statements-functions-classes-grammar model definition*). The parsed code includes all the relevant information for the analysis stage.

The analysis of the source code is done during the parsing phase as a validation mechanism. During the parsing, the user-defined analysis rule methods associated with the parsing rules of the grammar are applied in order to find out the violations at each grammar rule. The parsing engine executes the methods provided in the analysis rules associated with each grammar rule for identifying the violations, hence validating each analysis rule during the parsing of grammar rules. Within the checker rule methods, the user defines the checking mechanism for the rule using the target rule matching information and the state information. The violations are also saved within the methods by the user. At the end of the analysis, the violations are listed according to their user defined importance. The results of the analysis show how much the source code adheres to the coding rules defined in the checker, in other words the quality of the source code in terms of the analysis rules.

The grammar of the language, in GDML form, is managed by the user via the grammar editor of the user interface. All the necessary grammar sections and rules can be edited in this graphical interface. The analysis rules and their importance (error level), defined using CDML, are managed by the user using the checker rule editor of the user interface. The user manages the conditions to check in the grammar rules and associate analysis rules to the grammar rules of the language.

3.3.1. Code Document Object Model

Document Object Model (DOM) is a platform and language-neutral interface that permits scripts to access and update the content, structure, and style of a document. Programmers can create documents, explore their structure, and insert, change, or remove elements and content via DOM. On the other hand, CodeDOM is a language independent representation of source code. The source code is represented as a graph in CodeDOM, and this graph can be used for code generation.

Microsoft .NET Framework has the namespace *System.CodeDom* that supports CodeDOM. The namespace includes objects for representing, in a language independent fashion, most language structures. Using this namespace, one can generate source

codes in supported programming languages (C#, J#, VB.NET and Visual C++). *CodeDomProvider* can be used to create and retrieve instances of code generators and code compilers. Code generators can be used to generate source code for a particular language, and code compilers can be used to compile the source code into assemblies.

In this work, the *System.CodeDom* namespace of .NET Framework is used for generating, compiling and running the language specific parser according to the grammar and checker rules in the specified language.

3.4. Grammar Definition Language

Grammar Definition Language is the markup language that is used for defining a grammar of a language for the checker. We call this markup language as GDML (Grammar Definition Markup Language). GDML is based on XML. GDML comprises the necessary information for the checker to generate a parser for a specific language. The grammar of the language that includes the tokens and grammar rules of the language, and the methods for generating the parse tree of a source in the language are defined using GDML. In this section, GDML is discussed in detail.

Using a GDML definition for a specific programming language, the code checker generates the parser for that language and the generated parser is used for analyzing a given source code in that language according to a given checker rules definition for that GDML file. Therefore, the proposed flexible code checker can be used with different languages once their grammars are defined using GDML.

The syntax of the GDML file is depicted in figure 3.2. The figure shows the sample GDML definition for the C# language. GDML has the root element *Grammar* that defines the grammar of the language.

The root element *Grammar* has five attributes: *targetNamespace*, *targetClass*, *startRuleID*, *language* and *caseSensitive*. The attributes *targetNamespace* and *targetClass* define the namespace and the name of the generated language-specific parser

```
<?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"
targetNamespace="CSCodeParser"
targetClass="CSharpCodeParser"
startRuleID="CompilationUnit"
language="c#"
caseSensitive="true"
xmlns="http://www.cmpe.boun.edu.tr/grammar.xsd">
  <References>...</References>
  <Imports>...</Imports>
  <Keywords>...</Keywords>
  <TokenRegExs>...</TokenRegExs>
  <TokenTypes>...</TokenTypes>
  <StateDictionaries>...</StateDictionaries>
  <ValidatorFunctions>...</ValidatorFunctions>
  <ReplaceFunctions>...</ReplaceFunctions>
  <Rules>...</Rules>
</Grammar>
```

Figure 3.2. Sample GDML definition

class. The attribute *language* defines the language which is used in the code snippets in the parser, and should be one of the languages provided by CodeDOM Library (C#, VB, J#, Managed C++). The generated parser for the grammar is also generated in this language. The attribute *caseSensitive* denotes whether the language is case sensitive in terms of keywords and identifiers. On the other hand, the attribute *startRuleID* denotes the start rule of the grammar for parsing in the list of grammar rules and forms the root of the parse tree for that language.

The GDML tree consists of nine sections: *References*, *Imports*, *Keywords*, *TokenRegExs*, *TokenTypes*, *StateDictionaries*, *ValidatorFunctions*, *ReplaceFunctions*, and

Rules. Each section defines items that will be used in generating the parse tree of a given source code in the generated parser. The sections *Keywords*, *TokenRegExs*, *TokenTypes* constitute the tokenizer of the language for tokenizing a given source code while the other sections basically define the parser of the target language. Each of these sections is discussed in the following subsections.

There are some requirements that a GDML file has to satisfy for validity. First of all, all the items under the grammar sections, i.e. all tokens, rules, replace functions, must have non-empty and unique identifiers within a section. The keywords under the *Keywords* section must also be non-empty and unique within that section. The requirements of validity for each GDML section are summarized in the corresponding section below.

3.4.1. References

The References section of the GDML defines the list of assembly references to be used in the generated parser of the grammar. A project reference is by definition used to access and use the properties, methods, and events of a defined object. The items in the references section consist of valid assembly references that constitute the references of the automatically generated parser. A sample *References* section for the C# grammar that define the basic necessary references for the generated parser is given in figure 3.3.

3.4.2. Imports

The Imports section of the GDML defines the list of namespace imports to be used in the generated parser of the grammar. A namespace import is used for explicitly importing a namespace into a source file making all classes and interfaces of the imported namespace available. The items in this section consist of valid namespace imports that constitute the using namespace imports of the automatically generated parser for the target language. A sample *Imports* section for the C# grammar that define the basic necessary namespace imports is given in figure 3.4.

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

Figure 3.3. Sample GDML Reference definition

```
<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>SourceCodeParser</Import>
</Imports>
```

Figure 3.4. Sample GDML Import definition

3.4.3. Keywords

The *Keywords* section of the GDML defines the list of language specific keywords in the language of the grammar. The items in this section consist of the unique keywords of the target language that are used in the tokenization. The uniqueness depends on whether the language is case sensitive or not, which indeed is an attribute of the grammar. Part of keyword definitions in C# grammar is given as a sample *Keywords* definition in figure 3.5.

```
<Keywords>
  <Keyword>abstract</Keyword>
  <Keyword>as</Keyword>
  <Keyword>base</Keyword>
  <Keyword>bool</Keyword>
  <Keyword>break</Keyword>
  <Keyword>byte</Keyword>
  <Keyword>case</Keyword>
  <Keyword>catch</Keyword>
  <Keyword>char</Keyword>
  <Keyword>class</Keyword>
</Keywords>
```

Figure 3.5. Sample GDML Keyword definition

3.4.4. TokenRegExs

This section defines the list of token regular expressions in the language of the grammar. The token regular expressions are used for identifying the tokens during the tokenization phase. Token regular expressions are associated with token types for matching a token type.

A token regular expression consists of a unique identifier and a subsection called *Parts* that define the regular expression. A regular expression definition consists of a list of parts where each part is either a valid regular expression pattern or a valid reference to another token regular expression with its identifier. In the tokenization stage, the parts of the regular expressions are combined to obtain a single regular expression for matching the tokens.

Part of token regular expression definitions in C# grammar is given as a sample *TokenRegExs* definition in figure 3.6.

```
<TokenRegExs>
  <RegEx ID="NewLine">
    <Parts>
      <Part>(\r\n?|\n)</Part>
    </Parts>
  </RegEx>
  <RegEx ID="WhiteSpace">
    <Parts>
      <Part>[ \t]</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>
</TokenRegExs>
```

Figure 3.6. Sample GDML TokenRegex definition

The sample token regular expression definition in figure 3.6 contains four regular expression definitions. The first three regular expressions consist of single part that defines the regular expression of the item. For instance the regular expression corre-

sponding to the item *WhiteSpace* is: `[ \t]`. On the other hand, the fourth regular expression definition *EOLComment* , that defines the regular expression for matching single line code comments in C#, consists of five parts. Three of those parts are simple parts that contain a regular expression. The other two parts contain the attribute ID for referring to another regular expression with its identifier.

During the tokenization phase, the parts of each regular expression definition are combined to obtain a single regular expression for matching a token using that regular expression. Therefore the *EOLComment* regular expression definition in figure 3.6 corresponds to the regular expression: `(//[^\r\n]*(\r\n?|\n))` for matching a single line (end of line) code comment as a token.

3.4.5. TokenTypes

The section *TokenTypes* defines the list of token types in the language of the grammar. A token is the building block used in the tokenization phase. A token type is defined and matched using a valid regular expression definition.

A token type definition in GDML consists of a unique identifier, a regular expression reference called as *RegExID*, an optional flag called as *IsSignificant* and three optional subsections called *Summary*, *Example* and *Validator*. The regular expression reference must be the valid identifier of the regular expression definition associated with the token type that was defined in the *TokenRegExs* section.

The optional flag *IsSignificant* indicates whether the token type has significance in the parsing and execution of source codes in that language. For instance, comments and whitespace characters in C or C# are not significant in source files for compilation; they are mainly used for improving the readability of source files. Hence these token types should be defined as non-significant. Non-significant tokens are ignored in the parsing phase of the parser unless they are explicitly used in parsing for analysis. Non-significant tokens can be explicitly used grammar rules for matching them in the parsing phase and using them in checking the source code. For instance comments

can be used in grammar rules for checking the comment usage in a given source file in checker rules. The default value for the attribute *IsSignificant* is true; hence when it is omitted the token type is treated as significant.

The optional sections *Summary* and *Example* can be used to describe the token type and to specify an example for the token type definition respectively. The optional section *Validator* can be used to specify a method to post-validate the match of the token type. The method for the validator should be written in the language specified in the *language* attribute of the grammar. The method has the prototype given in the delegate definition in figure 3.7.

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

Figure 3.7. Token validator method delegate

The *buffer* parameter in the validator method definition is the content of the given source code being tokenized. The *offset* parameter of the method definition denotes the offset of the token match in the parameter *buffer*. The parameter *tokenString* is the matched token string at position *offset* in *buffer*. The validator method should validate the match and return *true* if the match is accepted and *false* otherwise.

Part of token type definitions in C# grammar is given as a sample *TokenTypes* definition in figure 3.8.

The sample token type definition in figure 3.8 contains four token type definitions. The first two token types identified as *Keyword* and *Identifier* use the same regular expression *Identifier*. In order to differentiate the token keyword from the token identifier, a validator method is defined that looks up the matched word from the dictionary of keywords defined in the *Keywords*. If the matched word does not match

```
<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="EOLComment" RegExID="EOLComment" IsSignificant="false">
  <Summary>End of line comment</Summary>
  <Example>// ...</Example>
</TokenType>
<TokenType ID="NewLine" RegExID="NewLine" IsSignificant="false">
  <Summary>Line break</Summary>
</TokenType>
```

Figure 3.8. Sample GDML TokenType definition

a keyword, then the word is matched as the token *Identifier*. The last two token types *EOLComment* and *NewLine* have the attribute *IsSignificant* set to false, since the end of line comment and newline are non-significant in the parsing phase.

3.4.6. StateDictionaries

The *StateDictionaries* section of the GDML defines the list of names of the state dictionaries that will be used to store the state information during the parsing phase. The state information is tracked in the parser state object during the parsing and analysis phases.

The state information consists of the list of dictionary items identified by the names defined in the *StateDictionaries* section, a single dictionary item for holding custom named item objects and the list of log items containing the objects logged

during the parsing and analysis.

The state information is necessary to differentiate a local variable from a global one or a parameter, or to identify the type or scope of a variable in grammar rules during the parsing phase. The state information is useful in the parsing and code checking phases to determine the current state of the parser, execute accordingly and update the state.

A dictionary item is a hash table accessed by its name holding state objects of any type. The modifications on dictionary items on the state are valid during the lifetime of the grammar rule modifying the item. When the rule and its siblings in the parse tree are matched successfully, the modifications made by these rules are reset by the parent rule of these rules. Hence the modifications on state dictionary items are valid through the siblings and children of a rule in the parse tree; they are not valid for the parent of the rule.

On the other hand, log items are used for logging information during the parsing and analysis phases. The modifications on the log items are permanent; they are valid throughout the parsing and analysis. The results of the analysis, the errors found in the analysis phase by the checker rules of the checker, are logged in the log items during the analysis.

State information is used and updated in the *Replace* methods and *Replaced* events of the rules to keep track of the parser's state. Part of state dictionary name definitions in C# grammar is given as a sample *StateDictionaries* definition in figure 3.9.

The sample state dictionary name definitions in figure 3.9 contains two dictionary name definitions. Hence, the list of dictionary items for this sample grammar consists of two dictionaries identified by names defined. Each of these dictionary items can be used for a specific purpose, such as list of currently defined variables, in the parser state.

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

Figure 3.9. Sample GDML StateDictionary definition

3.4.7. ValidatorFunctions

The *ValidatorFunctions* section of GDML defines a list of functions for validating the matches in complex grammar rules whenever necessary. These methods can be executed with their name identifiers by the grammar rules in their *Validator* sections for post validating a match. If the conditions validated by this method are not satisfied and the method returns *false*, then the match of the rule fails. These methods use the state information and the result of the rule match for validating the matches.

A validator function definition in GDML is identified by the *Function* element and has the *ID* attribute for identifying the method and referencing the method for validation in grammar rules. The method is defined under the *Code* section of the *Function* using the language specified in the *language* attribute of the grammar. The method has the prototype given in the delegate definition in figure 3.10 and should define a valid function body under this prototype.

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

Figure 3.10. Match validator method delegate

The *state* parameter in the validator method definition is the current state information of the parser to be used in the validation. The parameter *match* is the result object containing the match information, in the other words the list of items matched.

The validator method should validate the match and return *true* if the match is accepted. Otherwise it should return *false* and the match will be cancelled.

```
<ValidatorFunctions>
  <Function ID="OptionalPostCommaValidator">
    <Code>
      // "," can be used only in non-empty list
      return match.ItemCount(2) == 0 || match.ItemCount(1) == 1;
    </Code>
  </Function>
</ValidatorFunctions>
```

Figure 3.11. Sample GDML ValidatorFunction definition

A sample *ValidatorFunctions* definition from the C# grammar is given in figure 3.11. The sample validator method validates the result of the match by checking the condition that the optional third match item (a comma) should only exist whenever the optional second match item (first array item of the match result object) exists in a rule. This method can be used in grammar rules with its identifier for validating the condition that the optional comma should only exist when the optional list exists in matching comma seperated lists.

3.4.8. ReplaceFunctions

The *ReplaceFunctions* section of GDML defines the list of reusable replace functions for transforming the matches in grammar rules into higher level objects in the parsing phase of the code checker. These methods can be executed with their name identifiers by the grammar rules in their *Replace* sections for replacing matched items into match objects in the parse tree. These methods use and update the state information of the parser and use the result of the rule match for replacing the matched items into higher level objects in the parser. These methods return the replacement object for the match.

The returned replacement object of these methods represents the node of the rule in the parse tree in the generated parser, while the match items of the rule represents the children of this node in the tree.

During the parsing phase, the replace methods of the rules are executed when a rule is matched and matched items are removed from the match result. The replacement object returned from the replace method of the rule is added to the match result instead. Hence at each rule match, matched items are replaced with the replacement object returned from the replace method of the rule.

A replace function definition in GDML is identified by the *Function* element and has the *ID* attribute for identifying the method and using the method for replacement in grammar rules. The method body is defined under the *Code* section just like the validator functions. The method has the prototype given in the delegate definition in figure 3.12 and should define a valid function body under this prototype that returns the replacement object.

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

Figure 3.12. Match replace method delegate

The *state* parameter in the replace method definition is the current state information of the parser to be used and updated. For instance, in a variable declaration rule replace method, the parser should determine the scope of the defined variable using the current state information and then may add the newly declared variable's information into the state. The parameter *match* represents the list of matched items. The replace method must return the replacement object for the match that represents the matched items.

Part of replace function definitions in C# grammar is given as a sample *Replace-*

```
<ReplaceFunctions>
  <Function ID="RemoveMatch">
    <Code>
      // The function that will remove the match
      return null;
    </Code>
  </Function>
  <Function ID="ForwardItem0">
    <Code>return match.Item(0);</Code>
  </Function>
  <Function ID="SeperatedList">
    <Code>
      // match of the form "item(,item)*".
      // Return the items as an array
      int count = match.ItemCount(1) + 1;
      object[] list = new object[count];
      list[0] = match.Item<object>(0);
      for (int i = 1; i < count; i++)
        list[i] = match.Item<object>(1, i - 1);
      return list;
    </Code>
  </Function>
</ReplaceFunctions>
```

Figure 3.13. Sample GDML ReplaceFunction definition

Functions definition in figure 3.13. The first sample method, named as *RemoveMatch*, simply returns null value as a replacement, which causes the removal of the matched items from the match result as an effect. This method can be used for unsupported constructs in the target language, removing those constructs from the result. The second method *ForwardItem0* returns the first item in the match items, causing the list of

matched items to be replaced by the first matched item. On the other hand, the third method *SeperatedList* can be used to replace the items in the form of a list separated by commas into an array. The method returns the list as an *object* array.

3.4.9. Rules

The *Rules* section of GDML defines the list of grammar rules of the language. A rule represents the transformations of grammar items in the grammar. The grammar rules represent the grammar of the target language starting from the start rule, represented in the *startRuleID* attribute of the grammar. The grammar of the language should be represented in BNF notation. The tokens represent the terminal or leaf nodes of the grammar, while the start rule represents the root node.

A rule is identified by its *ID* attribute. A rule definition in GDML consists of an attribute named *matchType* and four sections called *Summary*, *Match*, *Replace* and *Validator*. The optional *Summary* section can be used to describe the rule.

The attribute *matchType* represents the type of the match of the rule. The possible values for the match type are *Sequence* and *Any*. The default value of match type is *Sequence*, hence when the match type is omitted it is treated as *Sequence*. The *Sequence* type represents a rule that matches a sequence of rule items and requires all of the match rule items to be matched for a successful match. Hence, it behaves like a logical and operator for the rule items. The *Any* match type represents a rule that matches any one of the rule items of the rule and requires any one of the match rule items to be matched for a successful match. Hence, it behaves like a logical or operator for the match rule items. In any mode, when any one of the rule items matches successfully, the rule itself also succeeds without continuing with the other rule items. Only in the case of a backtrack, the rest of the rule items can be executed.

Backtracking in matching rules of the grammar may occur when a rule matches some rule items successfully but the matching of that rule subsequently fails in the other rule items or the siblings of this rule. In this case, the parser engine tries to

backtrack in the matched items, if possible, and tries to find a successful match. In the case of a backtrack, all the effects of backtracked matches, including match result updates and state updates (including log items), has to be undone to prevent side effects.

The *Match* section is used for defining the rule items of the rule for matching the rule. The rule must define the list of rule items for defining how to match with that rule according to the match type of the rule. There are two types of rule items: *Recursive* and *Token*. The rule items have the common attribute *Quantity* which specifies the quantity of the rule item to match. The possible values for the *Quantity* attribute are as follows:

- ? (*ZeroOrOne*) : Optional, match zero or one time
- 1 (*ExactlyOne*) : Match exactly once
- * (*ZeroOrMore*) : Match any number of times (as many as possible)
- + (*OneOrMore*) : Match one or more times (as many as possible)

The default value for the quantity is *ExactlyOne*. The combination of rule items and match type specifies how to match that rule.

The *Recursive* rule item is used for referencing other rules with their identifiers. The *Recursive* rule items have the attribute *ID* to specify the referenced rule. The *Token* rule item is used for referencing token types with their identifiers. The *Token* rule items have the attributes *tokenType* and *token*. The *tokenType* attribute specifies the referenced token type. The optional *token* attribute can be used to specify a value for matching a token. The rule items have to specify valid references to rules and tokens to build a valid rule match list.

The optional *Validator* section can be used to specify a method to post validate the match of the rule. Using a validator method, matches that do not fulfill the required conditions can be filtered. In the validator section, either an inline method is defined for validating the match or predefined validator function, defined in the

section *ValidatorFunctions*, is referred with its identifier using the *ID* attribute of the validator function. The inline validator method has the same prototype with the reusable validator functions given in the delegate definition in figure 3.10.

The *Replace* section of the rule is necessary to specify a method to define a transformation of matched items of a successfully matched rule into higher level objects in the parse tree. In the replace section, similar to the validator section, either an inline replace method is defined for constructing a replacement object for the list of matched items or a method defined in the grammar section *ReplaceFunctions* is referred with its identifier using the *ID* attribute. The inline replace method has the same prototype with the reusable replace functions given in the delegate definition in figure 3.12. Each rule must have a replace method, either inline or referred, that returns the replacement object that replaces the list of matched items in the match result. The replace methods use and update the state information during this operation.

A valid rule should have all the references to other grammar items valid; rules, token types, replace functions and validator functions. The inline replace or validator methods defined should have a valid function code under its prototype. A rule should not have both an inline and referred function at the same time. Furthermore, in the *Any* match type, the rule item quantities * and + are not allowed, since this may cause an empty match.

Sample rule definitions from the rule definitions in C# grammar are given in figure 3.14 and figure 3.15. The first rule in figure 3.14, *ExpressionList*, consists of two match items (in sequence mode, both items have to match). Both items are recursive items referring to other rules and the second one has the quantity * (match zero or more times). The rule matches a list of expressions separated by commas using the rule items and returns the list of matched items as an array using the reusable replace function *SeperatedList* defined in the section *ReplaceFunctions*.

The second rule *ThisAccess* has a single match item of type token, which is a keyword token and has the value *this*. The rule has an inline replace method that

```
<Rule ID="ExpressionList">
  <Match>
    <Recursive ID="Expression" />
    <Recursive quantity="*" ID="ExpressionListRecursive" />
  </Match>
  <Replace ID="SeperatedList" />
</Rule>
<Rule ID="ThisAccess">
  <Match>
    <Token tokenType="Keyword" token="this"/>
  </Match>
  <Replace>
    return new ThisReferenceExpression();
  </Replace>
</Rule>
<Rule ID="BaseAccess" matchType="any">
  <Match>
    <Recursive ID="BaseAccessMember" />
    <Recursive ID="BaseAccessIndexer" />
  </Match>
  <Replace ID="ForwardObject" />
</Rule>
```

Figure 3.14. Sample GDML Rule definition 1

returns 'this reference' as an expression as the replacement object using the constructs in the logical layer. The third rule is an example of an *Any* type of rule. It contains two match rule items and it succeeds when any one of these items match. The method returns the replacement object of the rule item that has matched by referencing a replace function.

The sample rule given in figure 3.15, *ArrayInitializer*, consists of a summary, four

```
<Rule ID="ArrayInitializer">
  <Summary>Returns the array initializer as a list</Summary>
  <Match>
    <Token tokenType="Operator" token="{" />
    <Recursive quantity="?" ID="VariableInitializerList" />
    <Token quantity="?" tokenType="Operator" token="," />
    <Token tokenType="Operator" token="}" />
  </Match>
  <Validator ID="OptionalPostCommaValidator" />
  <Replace ID="ForwardItem1" />
</Rule>
```

Figure 3.15. Sample GDML Rule definition 2

match items, a referred validator method and a referred replace method. The second and the third match rule items have the match quanitiy ?, and are both optional. The validator method validates this case; the third match item (comma) should exist when the second match item (initializer list) exists. The rule returns the replacement of the second rule item (item 1) as a replacement.

3.5. Checker Rule Definition Language

Checker Rule Definition Language is the markup language that is used for defining the rules of the checker. We call this markup language as CDML (Checker rule Definition Markup Language). CDML is based on XML, similar to the syntax of GDML. CDML consists of the rule method definitions for the checker that is to be executed after the replace methods of the grammar rules as the replaced events of the grammar rules. Thus each checker rule definition is associated with a grammar rule. The rules of the checker are defined using CDML.

A CDML definition is associated with a GDML grammar, and the checker rules that are used for analyzing a given source code are associated with the grammar rules

of this grammar. Therefore, multiple CDML definitions for a single GDML grammar are possible, providing different checker rule bases to be used on the same grammar, hence the same language.

Using a CDML definition targeting the grammar of a target programming language, the code checker generates the checker for that language. The generated checker class is used in association with the target grammar's parser class for analyzing a given source code in that language. The checker rules are applied during the parsing phase of the parser.

The syntax of the CDML file is depicted in figure 3.16. The figure shows a sample CDML definition for the C# language grammar. CDML has the root element *Checker* that defines the checker rules.

The root element *Checker* has four attributes and three sections. The attributes *targetNamespace* and *targetClass* define the namespace and the name of the generated checker class. The attribute *language* defines the language which is used in the code snippets in the checker for defining checker rules, similar to the case of GDML. The generated checker class is generated in this language.

The attribute *targetGrammar* specifies the filename of the target grammar to use for the defined checker. The checker is associated with the target grammar and the checker rules are associated with the rules of the target grammar. Hence, a CDML definition defines the checker and specifies the target grammar.

CDML tree consists of three sections: *References*, *Imports*, *Rules*. The References section of the CDML defines the list of assembly references of the generated checker class. The items in the references section consist of valid assembly references as in the case of a GDML definition. Similarly, the Imports section of the CDML defines the list of namespace imports to be used in the generated checker class. The syntax of the References and Imports sections of CDML are same with that of GDML. Hence, the samples given in figure 3.3 and figure 3.4 are also valid for a sample CDML definition.

```
<?xml version="1.0" encoding="utf-8"?>
<Checker xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="CSCodeChecker"
  targetClass="CSharpCodeChecker"
  targetGrammar="D:\Grammars\CSharpCodeParser.xml"
  language="c#"
  xmlns="http://www.cmpe.boun.edu.tr/checker.xsd">
  <References>...</References>
  <Imports>...</Imports>
  <Rules>
    <Rule ID="MethodCommentCheck"
      targetRule="MethodDeclaration"
      importance="high">
      <Checker>
        if (e.Match.ItemCount(0) == 0)
          e.Log("Missing method comment");
      </Checker>
    </Rule>
  </Rules>
</Checker>
```

Figure 3.16. Sample CDML definition

The Rules section of CDML defines the rules of the checker. The rules specify the custom checking methods for the code checker. The code checker rules are defined with the *Rule* tag under the Rules section. A checker rule has the following three attributes:

- **ID** : Identifier of the checker rule.
- **targetRule** : The target grammar rule in the target grammar which the checker rule is associated with.
- **importance** : The importance (severity) level of the checker rule in terms of

analysis. The possible values for the importance are: *lowest*, *low*, *medium*, *high*, *highest*. The default value for the importance is *medium*.

The checker rule function is defined under the *Checker* section of the rule. There is also an optional section *Summary* for describing the rule. The checker rules are executed as events in the parser during parsing. These events correspond to a *Replaced* event for the GDML rules since they are executed after replace methods of GDML grammar rules. The *Rule Replaced* event has the prototype given in the delegate definition in figure 3.17.

```
public delegate void RuleReplacedEventHandler(
  object sender,
  RuleReplacedEventArgs e);
```

Figure 3.17. Rule replaced event delegate

The checker method defined in the *Checker* section is the handler of the *Replaced* event of the target grammar rule. The method uses the event's arguments as input, performs the check operation and logs the violations found in the state as log items. The event arguments are given as the class type *RuleReplacedEventArgs* that has the following properties:

- **State** : Current state information of parser.
- **Match** : The list of matched items of the target grammar rule.
- **Replacement** : The replacement object returned from the replace method of the grammar rule.
- **CheckerRule** : The checker rule object being executed.
- **StartLine** : The line number of the first token in the source code that matches the target grammar rule (Starting line number of match).
- **EndLine** : The line number of the last token in the source code that matches the target grammar rule (Ending line number of match).

The checker rules use the state information, matched rule items in the target grammar rule and the replacement object of the target grammar rule for performing the analysis. The errors found in the analysis are added to the log items of the state so that they are preserved during the parsing and they can be accessed at the end of the parsing. The checker errors are added to the log of the state as a specific *CheckerError* object so that they can be listed conveniently as the result of the check. The *CheckerError* class has the following members:

- **CheckerRule** : The current checker rule object being executed.
- **ErrorMessage** : The specific error message for the error.
- **LineNumber** : The line number to navigate for the error in the source code.

A checker error can simply be logged in the state using the *Log* function of the *RuleReplacedEventArgs* object. The *Log* method gets the error message as the parameter, and defines a new *CheckerError* using the information in the event arguments and the error message. The error message specifies the specific error message to be displayed for that specific checker error. The *Log* method uses the *StartLine* property of the event arguments for determining the line number and uses the starting line number of the matched rule as the *LineNumber*. The user defines a checker error in the log items of the state using the *Log* method and the list of errors are displayed at the end of the analysis to the user in the form of (importance, rule ID, error message, line number) according to the importance of the rules.

The sample CDML definition given in figure 3.16, contains a checker rule, named as *MethodCommentCheck*, that checks the existence of a comment before a method definition. If the optional first match item for the target grammar rule, *MethodDeclaration*, is not matched (the item count is zero); then the matched method definition does not have a comment before the method and this constitutes an error in the checker. The rule has the importance *high*, specifying the error importance level for the rule. The found error is registered in the log items of the state using the *Log* method of the event arguments with the error message.

There are some requirements that a CDML definition has to satisfy for validity. First of all, a CDML definition must have a valid target grammar definition in order to define a checker for that grammar. The checker rules must have unique identifiers and must have valid target grammar rules. Furthermore, the checker rule methods must define valid method bodies according to their prototype.

3.6. Implementation

The designed source code checker is implemented in C# by using Microsoft.NET Framework. The implementation can be divided into five logical parts:

- Logical Layer
- GDML and CDML Deserialization
- Parser and Checker Generation
- Code Analysis
- User Interface

In the Logical Layer, the building blocks constituting the parse tree during the parsing and analysis of the checker are defined. These building blocks are used in transformation of match rule items to higher levels in the parse tree in the rule replace methods.

GDML and CDML Deserialization phase is used for deserializing GDML and CDML content into specific code checker classes. A given GDML definition is deserialized into the *Grammar* class that contains all the GDML content. A CDML definition is on the other hand is deserialized into the *CheckerDefinition* class.

The Parser and Checker Generation phase is used for generating the language specific parser and checker for the target language according to its grammar and code checker definitions. This phase provides the functionality for generating a parser class and a checker class for the target language, that extend the language independent base parser and checker classes and that contain all definitions in the given GDML and

CDML contents respectively.

The Code Analyzer layer contains the necessary classes and methods for checking a given source code using the generated parser and checker classes. A given source code is parsed according to the generated parser, and the checker rules defined in the generated checker class are executed during the parsing for finding out the corresponding violations.

On the other hand, the User Interface layer comprises the high level graphical user interfaces that interfere with the user and provides the interface for the checker by making use of the methods in the underlying analysis layer.

The implementation details of these five phases are discussed in the following subsections.

3.6.1. Logical Layer

The Logical Layer of the code checker defines the building blocks of the generated parse tree in parsing a given source code. The parse tree consists of the tokens of the tokenized source code at leaf nodes and logical layer items in the other nodes. Hence, the logical layer defines the objects for the transformation of lower level parse tree objects into a higher level object. These objects are used in the parsing and analysis phases of the checker. The replace methods of grammar rules and checker rule methods use these building blocks. The replace methods of grammar rules use the logical layer objects for defining the replacement object for the match items of a rule.

In the logical layer, language independent constructs representing generic constructs in programming languages such as expressions, statements, methods etc. are defined. The defined constructs are similar to the CodeDOM structures for representing language specific constructs but also includes some constructs that are not included in CodeDOM library. The highest level building blocks in the logical layer are structural definition blocks such as model definition, class definition, method definition etc.

On the other hand lower level building blocks in the logical layer consist of operational building blocks in a programming language such as statements and expressions. The expressions form the lowest level building blocks in the logical layer and they reside above the tokens, which form the leaf nodes, in the parse tree. Each building block represents a language construct and contains the necessary fields for representing the specific construct. The class diagrams of the logical layer classes are given in the appendix.

Structural definition blocks in the logical layer include namespace declarations, delegate declarations, class declarations, event declarations, field declarations, property declarations and method declarations for modeling these definitions. The classes representing the structural definition blocks are summarized in table 3.1.

The operational definition blocks in the logical layer consist of class definitions for representing operational language constructs in programming languages. The operational constructs consist of statements and the statements consist of expressions and tokens. The classes representing the operational definition blocks are summarized in table 3.2.

Statement class is the basic building block for representing the statements in a language. It is extended by statement type classes representing different statement types in programming languages. Each statement class represents a different type of a statement and contains different members for this purpose. The statement type classes are summarized in table 3.3. The *Statement* class contains the following utility functions:

- **EmptyLineCount** : Number of blank lines in the statement (extended by statement type classes).
- **IsSpecificStatement** : Check whether the statement is of a specific type given in the predicate delegate method as the parameter (not extended).
- **StatementCount** : Number of effective (significant, non-whitespace) statements in the statement (extended by statement type classes).

Table 3.1. Logical Layer structural definition classes

Class Name	**Explanation**
ModelDefinition	Stores project related information and list of namespace definitions
NamespaceDefinition	Stores namespace information such as name, imported namespaces, and lists of class, delegate, enum, struct and interface definitions
DelegateDefinition	Stores delegate information such as name, return type and parameter definitions
ClassDefinition	Stores class information such as name, access modifiers and list of member definitions
StructDefinition	Stores struct information such as name, access modifiers and list of member definitions
EnumDefinition	Stores enum information such as name, access modifiers and list of member definitions
EventDefinition	Stores event information such as type, name and access modifiers
FieldDefinition	Stores field information such as type, name, access mode and optional initialization value
PropertyDefinition	Stores property information such as name, return type, access modifiers and get/set method definitions
MethodDefinition	Represents a method definition and stores method information such as method name, return type, access modifiers, list of parameter definitions and list of statements of the method body
ParameterDefinition	Represents a parameter definition and stores parameter information such as name, type and access modifier
InterfaceDefinition	Stores interface information such as name, access modifiers and list of member definitions
AttributeDefinition	Stores attribute information such as name and attribute value list

Table 3.2. Logical Layer operational definition classes

Class Name	**Explanation**
Statement	Base class for representing a statement
StatementCollection	Represents a list of statements, used to model statement blocks
Expression	Base class for representing an expression

These utility functions are extended by necessary statement type classes (that include blocks as statement collections) for obtaining the desired behavior. For instance, the StatementCount method returns one by default in the Statement class, but the WhileStatement class extends the method and returns the number of statements in the while block (by recursively invoking the method) plus one.

On the other hand, *StatementCollection* class represents a list of statements and is used for representing statement blocks consisting of a list of statements. It is used within language constructs containing statement blocks such as methods or statements (i.e. if statement, while statement etc.). It contains the following utility functions for working with statement blocks:

- **EmptyLineCount** : Total number of blank lines in the statement collection.
- **StatementCount** : Total number of effective statements in the statement collection.
- **GetStatement** : Get a specific statement in the statement collection with an index.
- **LastStatement** : Get the last statement in the statement collection.

Expression class represents the smallest building blocks other than tokens in a language. Statements are composed of tokens and expressions in the parse tree. The Expression class is extended by expression type classes representing different types of expressions in programming languages. The expression type classes are summarized in table 3.4 and table 3.5. The *Expression* class contains the following utility functions that are extended by necessary expression type classes:

Table 3.3. Logical Layer statement types

Class Name	**Members**
AssignStatement	leftExpression, rightExpression
AttachEventStatement	eventReference(Expression), listener(Expression)
RemoveEventStatement	eventReference(Expression), listener(Expression)
BreakStatement	-
ContinueStatement	-
CommentStatement	text
WhiteSpaceStatement	element
ConditionStatement	condition, trueBlock, falseBlock
SwitchStatement	switchExpression, switchBlock (section list)
ExpressionStatement	expression
GotoStatement	label
LabeledStatement	label, statement
IterationForStatement	initStatements, testExpression, incerementStatements, iterationBlock
IterationWhileStatement	testExpression, iterationBlock
IterationDoWhileStatement	testExpression, iterationBlock
IterationForeachStatement	variableType,varName,loopExpr,iterationBlock
MethodReturnStatement	expression
ThrowExceptionStatement	toThrow (Expression)
CatchClause	name, exceptionType, catchBlock
TryCatchFinallyStatement	tryBlock, catchClauses, finallyBlock
UsingStatement	acquisition, usingBlock
VariableDeclarationStatement	name, type, initExpression

- **ExpressionCount** : Number of subexpressions of the expression.
- **ContainsExpression** : Check whether the expression contains a specific expression type given in the predicate.
- **ContainsBinaryOperator** : Check whether the expression contains a specific binary operator.

Table 3.4. Logical Layer expression types - 1

Class Name	**Explanation**
AddressOfExpression	Represents an address of reference
ArgumentReferenceExpression	Represents a parameter reference
ArrayCreateExpression	Represents an array creation
ArrayIndexerExpression	Represents an array index reference
BaseReferenceExpression	Represents base class reference
BinaryOperatorExpression	Represents a binary operator expression
CastExpression	Represents a cast to a specified type
DelegateCreateExpression	Represents a delegate creation
DelegateInvokeExpression	Represents a delegate invocation
DereferenceExpression	Represents a dereferencing of a pointer
DirectionExpression	Represents the direction of a parameter
EventReferenceExpression	Represents an event reference
FieldReferenceExpression	Represents a field reference
IndexerExpression	Represents an index reference
MethodInvokeExpression	Represents a method invocation
MethodReferenceExpression	Represents a method reference

A sample source code to parse tree transformation using the logical layer building blocks is depicted in figure 3.18 and figure 3.19. Figure 3.18 contains a sample source and figure 3.19 represents the corresponding parse tree.

```
public int Add (int a, int b)
{
  int sum = a + b;
  return sum;
}
```

Figure 3.18. Sample method for parse tree generation

Table 3.5. Logical Layer expression types - 2

ObjectCreateExpression	Represents a new instance creation or a pointer memory allocation
ObjectDeleteExpression	Represents a pointer memory deallocation
ParameterDeclarationExpression	Represents a parameter declaration
PreDecrementExpression	Represents a predecrement expression
PreIncrementExpression	Represents a preincrement expression
PostDecrementExpression	Represents a postdecrement expression
PostIncrementExpression	Represents a postincrement expression
PrimitiveExpression	Represents a primitive value
PropertyReferenceExpression	Represents a property reference
PropertySetValueReferenceExpression	Represents value argument of property
SizeofExpression	Represents a sizeof expression
SwitchDefaultCaseExpression	Represents default case of switch
ThisReferenceExpression	Represents current local class instance
TypeOfExpression	get type of an object
TypeReferenceExpression	Referencing a type
VariableReferenceExpression	Represents a variable reference
WhitespaceExpression	Represents a whitespace

The simple method definition in figure 3.18, represents a simple add function implementation in C#. The corresponding parse tree for this method according to the sample C# grammar is given in figure 3.19. The parse tree consists of the method definition at the root. The method definition consists of two parts, the header and the body. The header part matches the method information. The body is a statement collection and consists of two statements; a declaration statement, which declares an integer variable, and a method return statement.

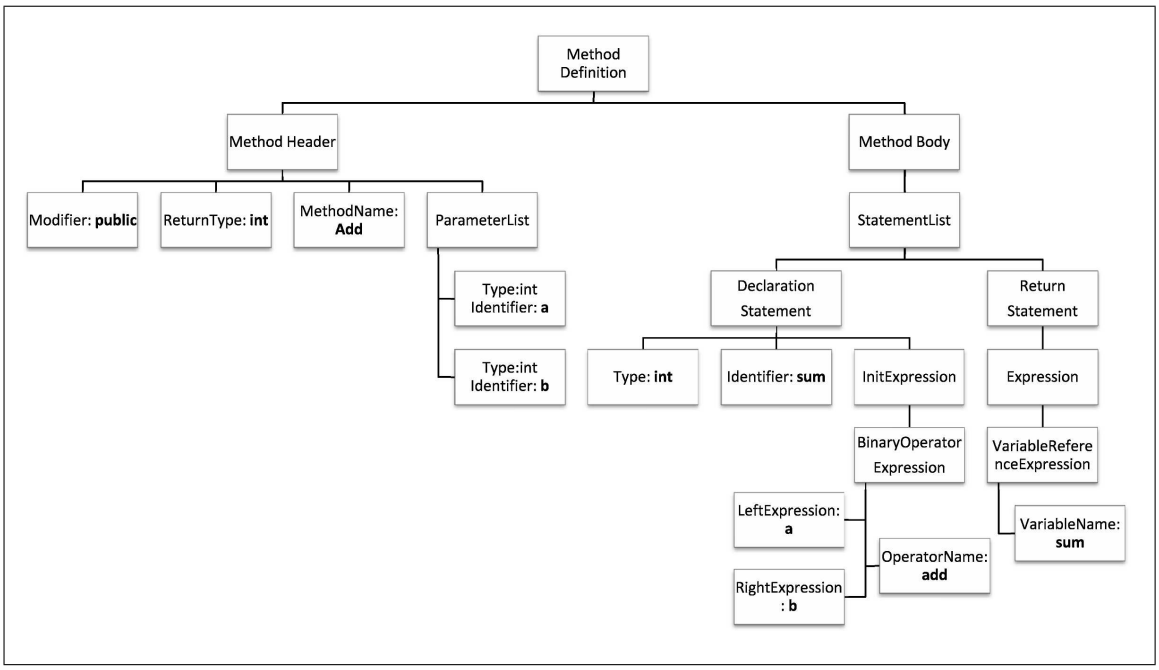

Figure 3.19. Parse tree for the sample source code

3.6.2. GDML and CDML Deserialization

The first step in checking a given source code is to prepare the language specific parser for the target language according to the given GDML grammar and CDML checker rules. Thus the first step is to deserialize given GDML and CDML definitions into code checker classes. In this section, the classes used in deserialization operation are discussed.

The GDML content is deserialized as an instance of the *Grammar* class. Hence, the *Grammar* class corresponds to the GDML definition and has the members for holding the information in the attributes and sections of the GDML grammar definition.

The Grammar class contains the following properties that correspond to the attributes and sections of the GDML grammar:

- *TargetNamespace*: Corresponds to the attribute *targetNamespace*, represents the namespace of the generated parser.
- *TargetClass*: Corresponds to the attribute *targetClass*, represents the name of

the main class of the generated parser.

- *Language*: Corresponds to the attribute *language*, represents the programming language used in method definitions and in generation of the parser.
- *StartRuleID*: Corresponds to the attribute *startRuleID*, marks the start rule of the generated parser.
- *CaseSensitive*: Corresponds to the attribute *caseSensitive*, represents whether the identifiers in the target language are case sensitive.
- *References*: Corresponds to the section *References*, represents the list of assembly references for the generated parser.
- *Imports*: Corresponds to the section *Imports*, represents the list of namespace imports for the generated parser.
- *Keywords*: Corresponds to the section *Keywords*, represents the list of keywords in the target language.
- *TokenRegExs*: Corresponds to the section *TokenRegExs*, represents the collection of regular expression definitions in GDML. Each token regular expression is mapped to the *XmlTokenRegEx* class, that represents a token regular expression definition.
- *TokenTypes*: Corresponds to the section *TokenTypes*, represents the collection of token types in GDML. Each token type definition is deserialized to the *XmlTokenType* class, that represents a token type definition.
- *StateDictionaries*: Corresponds to the section *StateDictionaries*, represents the list of state dictionary names for the generated parser.
- *ValidatorFunctions*: Corresponds to the section *ValidatorFunctions*, represents the collection of validator functions in GDML. Each validator function definition is mapped to the *XmlIdentifiableFunction* class, that represents a function definition.
- *ReplaceFunctions*: Corresponds to the section *ReplaceFunctions*, represents the collection of replace functions in GDML. Each replace function definition is mapped to the *XmlIdentifiableFunction* class, that represents a function definition.
- *Rules*: Corresponds to the section *Rules*, represents the collection of rules in GDML. Each grammar rule definition is deserialized to the *MatchRule* class,

that represents a grammar rule definition.

The *TokenRegExs*, *TokenTypes*, *ValidatorFunctions*, *ReplaceFunctions* and the *Rules* properties of the Grammar class are collections that are identified with an *ID* property, *IdentifiableElementCollection*. Each of these is mapped to their corresponding classes: *XmlTokenRegEx*, *XmlTokenType*, *XmlIdentifiableFunction* (for validator and replace functions) and *MatchRule*. Each of these classes extends the IdentifiableElementCollection class and contains the properties for mapping the corresponding GDML sections into the instances of these classes.

The *Grammar* class also contains the necessary methods for handling the serialization to and deserialization from grammar files, generation of the parser and validation of the grammar. The grammar is validated against the constraints defined in the GDML section 3.4 for the sections of the GDML.

On the other hand, the CDML content is deserialized as an instance of the *CheckerDefinition* class that corresponds to the CDML definition, and has the members for holding the information in the attributes and sections of the CDML checker rules definition like the *Grammar* class. The CheckerDefinition class contains the following properties that correspond to the attributes and sections of the CDML checker:

- *TargetGrammar*: Corresponds to the attribute *targetGrammar*, represents the filename of the target grammar.
- *TargetNamespace*: Corresponds to the attribute *targetNamespace*.
- *TargetClass*: Corresponds to the attribute *targetClass*.
- *Language*: Corresponds to the attribute *language*, represents the programming language used in checker rule method definitions and in generation of the checker.
- *Rules*: Corresponds to the section *Rules*, represents the collection of checker rules in CDML. Each checker rule definition is mapped to the *CheckerRule* class, that represents a checker rule definition.

The *CheckerDefinition* class also contains the necessary methods for handling

the serialization to and deserialization from checker rule base files and validation of the checker rules similar to that of the *Grammar* class.

3.6.3. Parser and Checker Generation

The Parser and Checker Generation phase is used for generating the language specific parser and checker classes for the target language according to the given grammar and code checker definitions. Once a checker definition is given, the grammar to use for that checker for parsing is identified from the target grammar of the checker.

The language specific parser and checker classes for the target language are generated when the GDML and CDML content are deserialized into a *Grammar* and *CheckerDefinition* instances. These classes are generated in the language specified in the *language* attribute of the GDML and CDML files respectively. These programming languages should be one of the programming languages supported by the CodeDOM library. These languages also specify the language for defining the code snippets for validation, replace methods in GDML and checker methods in CDML definitions respectively. They are not related with the language of the target grammar.

The generated parser and checker classes can be used for analyzing a given source code. Furthermore, these classes can be saved as text files for checking them and also as dynamic assemblies for packaging them. These assemblies can be used directly to analyze a given source code without needing to generate them every time for analysis.

There is a base parser class which is extended by the generated language specific parser class. This base class contains the language independent features of the parser such as the parse operation (using tokens, rules and methods). The same argument also holds for the checker class. The generated checker class extends the base checker class, which contains language independent features such as running the checker.

The generated classes have the namespace, class name, assembly references and namespace imports defined in the respective GDML and CDML definitions. Further-

more, they contain the items defined under the GDML and CDML definitions.

The token regular expressions, token types and rules are defined as subclasses in the generated parser. These subclasses contain the relevant information of the corresponding grammar section (TokenRegexs, TokenTypes and Rules). On the other hand the token validator methods, rule validator methods (ValidatorFunctions and inline validator methods), rule replace methods (ReplaceFunctions and inline replace methods) are defined as methods under the generated parser. The checker rule methods are defined as methods in the generated checker.

3.6.4. Code Analysis

This layer of the code checker is the operational part of the application and provides the necessary operations for code checking. It consists of the three stages mentioned in the design section: Tokenizer, Parser and Analyzer. It is responsible for the following tasks:

- Tokenization of the source code
- Parsing of the tokenized code
- Checking the parsed code according to the rules of the checker
- Saving and loading grammar definition bases for the checker
- Saving and loading checker rule definition bases for the checker

The first step in the code checker is to load the given checker definition and then using the target grammar of the checker definition to load the grammar for the target language into the checker definition and grammar instances. The next step is to generate the parser and checker classes as described in the previous sections 3.6.2 and 3.6.3. These operations are done using the functionalities provided in those sections. The code checker phase uses the generated parser and checker for performing the above tasks.

On the other hand, if the checker and grammar definitions are supplied as a previously saved assembly, the generation is not necessary since the parser or checker classes can be loaded directly from a valid assembly in this case.

In the tokenization part, the given source code is read and parsed into tokens of the target language using the token regular expressions and token types in the grammar of the language. A token is the simplest code segment in the language that can be identified. At the end of this part; a list of tokens,tokenized source, is obtained for the given source code.

The keywords of the target language are also used as a token type during the tokenization phase. The case sensitive flag of the grammar specifies whether the tokens should be matched as case sensitive or not.

The next phases in code checking are the parsing and the analysis phases. These phases are closely related and overlap since the code checking is also done during the parsing operation.

In the parse operation, the obtained list of tokens from the tokenization phase is parsed according to the grammar rules and functions obtained from the grammar of the target language into the generated parser. The parser of the code checker is a recursive descent parser. A recursive descent parser is defined as a top-down parser built from a set of recursive procedures where each procedure implements one of the production rules of the grammar. As a result, the structure of the resulting program closely mirrors that of the grammar it recognizes. Therefore, in the parsing phase we start with the list of tokens as the result object and they form the leaves of the parse tree of the source file. Then, starting with the start rule of the grammar, we start matching the rules on the list of tokens in a top-down manner.

For each rule during the matching of the rules, if the match type is *Sequence*, all of the rule match items have to match for a successful match of the rule. On the other hand, if the match type is *Any*, then a match of one of the rule items is enough for

the success of the rule. For each rule, rule items of the rule are successively applied trying to match the rule (until a rule item matches in any mode or a rule item fails in sequence mode).

For each rule item of rules; if the match rule item is a token, that token has to be matched in the current position in the token list for success. When a match rule item that refers to another rule item (*Recursive*) is encountered, the match function is recursively called with that rule at the current position in the token list and that rule tries to match. When a rule is called at the same token position in the second time, the second call fails in order to avoid an infinite recursion. In this way, the recursive calls will terminate upon reaching token items since the tokens form the leaves of the parse tree and the terminals of the BNF grammar of the language. The position in the token list is updated during the advancing in the match rule items with success in the *Sequence* mode or with successes of rules.

When a rule matches successfully, its validator method is executed for post validating the match whenever it exists. If it also succeeds, the rule succeeds and its replace method is executed. The list of matched items is removed from the result and the replacement object returned from the replace method is inserted instead. Hence, the rule completes a transformation from the list of matched items to the replacement in the result. After the replace method, the replaced event that is used for the checker rules is fired. If the rule has an associated checker rule, then the method of the checker rule is executed for checking the relevant rule and logging the errors found in the state.

When a rule item fails, the matching operation continues with the other rule items in Any mode. On the other hand, in Sequence mode or when all rule items fail in Any mode, the associated rule also fails and this effect is transmitted to the parent rule (the corresponding rule item running this rule also fails).

The state information is used and updated during the replace and checker rule methods for keeping track of the parser state. Furthermore, there is the possibility of backtracking in the rules. When a rule succeeds in some match items but then fails in

a succeeding match rule item, the parser backtracks to try to find another successful match whenever possible. In the case of backtracking, all modifications done by the backtracked rules are cancelled including the logged items in the state.

With the above operation methodology, the list of tokens is transformed into higher level structures (i.e. 2+3 is combined into an expression, x=2+3 is then combined into an assignment statement etc.). At the end, we end up with the parse tree of the source code for a valid source.

As a result we also obtain all the violations of the checker rules in the log items of the state. All the found defects are listed in the interface based on their checker rule importance. The violations together with their importance show the quality of the source code against the specified checker rules

3.6.5. User Interface

The user interface layer of the Flexible Code Checker includes the graphical user interfaces of the application. It is the front end of the source code checker application and presents the user the functionalities provided by the underlying analyzer layer. The user interface is responsible for the following tasks:

- Collecting the inputs from the user
- Providing the interface for executing the rule checker
- Displaying the violation of checker rules found in the source code
- Providing the interface for GDML grammar management
- Displaying the errors of validation of the grammar
- Providing the interface for CDML checker rules management
- Displaying the errors of validation of the checker rule base
- Providing the interface for loading, editing and saving language grammars
- Providing the interface for loading, editing and saving checker rule bases
- Providing the interface for loading, editing and saving project files

The user interface contains the forms for managing a GDML definition and a CDML definition. The user can build the grammar or the checker and validate them using these interfaces. Furthermore the execution form of the checker is used for running the checker on a target source file by selecting the CDML definition (or assembly) and the target source file. These files can be also be saved and loaded as project bases for later usage.

4. SAMPLE APPLICATIONS

In this chapter, sample applications built with the proposed flexible code checker framework are described. The developed flexible code checking system has been used to build code checkers for two programming languages: C# and C++. The grammars of the languages have been defined using GDML and sample checker rules for analyzing sample sources are defined using CDML. Then the checkers are tested on sample source codes for identifying the violations of the checker rules.

The details on the sample checkers are given in sections 4.1 and 4.2. Then in section 4.3; performance, accuracy and feasibility issues of the code checker framework are discussed. In section 4.4, the proposed checker framework is compared against some of the existing commercial tools.

4.1. Sample C# Checker

A sample source code checker for the C# language is implemented using the developed code checker framework in this study. The language constructs in the C# language are defined in the form of GDML for parsing given C# sources. Then the checker rules are defined in the form of CDML for analyzing given sources according to these rules.

The sample C# code checker has been the main testing platform for the flexible code checker in this study since C# grammar was the first grammar fully defined in GDML form.

In order to build the sample C# checker, the first step is to define the grammar of the C# language in GDML form and then define the necessary transformation functions in the grammar for building a parse tree using the grammar. The C# language grammar defined in the MSDN Library [20], has been used for defining the grammar. The root of the C# grammar, hence the start rule in the GDML definition, is defined in

the *CompilationUnit* rule of the grammar. The target grammar is mapped to a GDML definition using the GDML Editor of the code checker by defining the keywords, tokens, token regular expressions for matching tokens and grammar rules.

The target C# grammar is fully defined covering all of the language constructs in the language. Only unsafe language constructs in the target grammar are not covered for standardization. After representing the target grammar in GDML form, the replace functions of grammar rules are defined for transforming matched rule objects to higher level representations. The constructs in the logical layer of the code checker are used in the transformation process for mapping lower level parse tree objects to higher level constructs. For instance, the replace function of the grammar rule *MethodDeclaration* constructs and returns a *MethodDefinition* object of the logical layer by using the matched items in the rule: the method header and the method body. Therefore the matched method header and method body are transformed into a method definition in this rule. The preprocessing directives are not handled for simplification in this phase, and they are skipped during the parsing in the defined grammar.

Since the grammar of the C# language is quite complex and long, the definition of the target grammar is also a time consuming task. But once the grammar is defined, it can be reused in defining various checkers for the C# language.

After defining the target grammar in GDML form, a sample checker on this grammar is defined in CDML form by defining sample rules for the analysis. The sample checker contains various types of checker rules for analyzing a given source in C#. The sample checker rules are summarized in tables 4.1 and 4.2.

The sample checker rules contain various types of analysis rules ranging from naming and complexity rules to convention rules. The definition of these rules are not complex, since the necessary items for checking a match such as matched items, state information and replacement objects are available. The checker executes the rules defined in the checker and the items logged in the checker methods are collected as a result.

Table 4.1. Sample C# checker rules - 1

Rule Name	**Explanation**
MethodCommentCheck	Checks for a comment preceding a method declaration (for describing the method)
MethodLengthCheck	Checks the number of significant statements in a method body (should not be too high)
MethodEmptyLineRate	Checks the number of effective (non-empty) lines (should not be too high) and the rate of empty lines over the effective lines (should not be too low) in a method body.
MethodCommentRate	Checks the rate of comments over the total number of statements (should not be too low) in a method body
MethodParameterCount	Checks the number of parameters in a method declaration (should not be too high)
MethodIfCounter	Checks number of if statements (number of branches) in a method body
SwitchDefaultCheck	Checks for existence of the default case in switch statements
SwitchCaseBreakCheck	Checks for a break statement at the end of case blocks in switch statements

The behavior of the checker is completely defined in the checker rules, hence no false positives or false negatives will occur with properly defined checker rule methods. On the other hand, the complexity of the checker rule methods entirely depend on the implementation of the method and the complexity of the analysis rule. Some analysis rules may require complex checker rule method definitions or too much tracking in the parsing using the state, making the analysis on these items quite infeasible.

The sample checker is executed on different sample C# source files for analyzing them according to the defined checker rules. The violations of the checker rules are correctly identified and listed according to their importance.

Table 4.2. Sample C# checker rules - 2

Rule Name	**Explanation**
VariableNameCheck	Checks whether local integer variable names start with i and does not include _ in local variable declarations
VariableInitializerCheck	Checks whether local integer variables are initialized in local variable declarations
IfConditionCheck	Checks for assignment in conditions of if statements
WhileConditionCheck	Checks for assignment in conditions of while statements
ComplexExpressionCheck	Checks expressions for the number of subexpressions included (for detecting complex expressions)
ComplexExpPrePostInc	Checks usage of pre/post increment and decrement expressions other than stand-alone in expressions
MethodReturnCount	Checks number of return points of a method declaration (should not be more than a few)
MethodReturnCheck	Checks whether the final statement of a non-void method is a return statement
PrivateFieldNameCheck	Checks whether private field names start with _ in field declarations

4.2. Sample C++ Checker

A sample source code checker for the C++ language is implemented using the developed code checker framework to emphasize the flexibility of the checker on different target languages and checker bases. For this purpose, the language constructs in the C++ language are defined in the form of GDML and the checker rules are defined in the form of CDML for analyzing given sources according to these rules for the sample C++ checker.

In building the sample C++ checker, the first step is to define the grammar of the C++ language in GDML form and then define the necessary transformation functions

in the grammar for building a parse tree using the grammar. The C++ language grammar defined in the BNF form in [21], has been used for defining the grammar. The root of the C++ grammar, hence the start rule in the GDML definition, is defined in the *TranslationUnit* rule of the grammar, which defines a series of declarations. The target grammar is mapped to a GDML definition using the GDML Editor of the code checker by defining the keywords, tokens, token regular expressions for matching tokens and grammar rules.

In order to simplify the grammar definition, some language constructs are not implemented in the target grammar definition. The preprocessing directives and template constructs are not implemented in the grammar definition for this purpose. The preprocessing directives are marked as non-siginificant and are skipped during the parsing phase. After representing the target grammar in GDML form, the replace functions of the grammar rules are defined for transforming matched rule objects to higher level representations. The constructs in the logical layer of the code checker are used in the transformation process for mapping lower level parse tree objects to higher level constructs.

After defining the target grammar in GDML form, a sample checker on this grammar can be defined in CDML form by defining sample rules for the analysis. The sample checker for C++ uses some of the sample checker rules given in tables 4.1 and 4.2 by mapping them to the GDML definition of the C++ language. Furthermore, it also includes the rules summarized in table 4.3 that are specific for the C++ grammar.

The sample checker is executed on different sample C++ source files for analyzing them according to the defined checker rules. The violations of the checker rules are correctly identified according to the checker rule methods and listed according to their importance.

The CDML definition of the sample C++ checker and the results of a sample execution of the checker on a sample source file are given in the appendix.

Table 4.3. Sample C++ checker rules

Rule Name	**Explanation**
VariableNameCheck	Checks whether local integer variable names does not start with _ in local variable declarations
PtrVariableInitializerCheck	Checks whether pointer variables are initialized in declarations
PrivateFieldNameCheck	Checks whether private field names in classes start with _ in field declarations and usage of non-private fields in classes
GlobalVariableDeclaration	Checks for global variable declarations
GlobalVariableModification	Checks for modification of global variables in methods as a side effect
GlobalVariableUsage	Checks for usage of each global variable in methods

4.3. Performance Issues

The performance of a checker that is built using the code checker framework depends on the user defined methods in the checker. The factors affecting the performance of the checker can be summarized as follows:

- Complexity of language independent internal parser.
- Complexity of target grammar.
- Complexity of user-defined replacement methods in grammar definition.
- Complexity of user-defined checker methods in checker definition.

Only the complexity of the first item, the complexity of the internal language independent parser is fixed. The other items depend on the specific target language grammar and user-defined methods in the checker. The complexity of the first item is mostly dominated by the complexity of the other items. Hence, the complexity of a checker depends on the complexity of the target language and definition of the checker via the complexity of the grammar definition, the complexity of the replacement meth-

ods and the complexity of checker rule methods. The definition of the target grammar via the order of rule items or the amount of recursions also affect the performance.

On the other hand, the accuracy of the checker is also user-defined and it depends on the accuracy of the user defined checker methods. The behavior of a checker is defined in the checker methods and the accuracy of this observed behavior against the desired behavior depends on the checker methods. There will be no false positives and false negatives in an appropriately defined checker.

Defining a grammar definition for a target language in the checker may be complex due to the complexity of the grammar of the target language. In fact, it is the most difficult part in building a new checker for a target language using the framework. On the other hand, once the grammar is defined, the grammar definition can be used with different checker rule bases.

In general, the checker rule method definitions are not too complex. But in some cases, building a checker rule for analyzing a certain behavior may require too much tracking in the grammar and state, and may become too complex to implement. Hence, it will not be feasible and worth to define that checker rule for analyzing that behavior in this case.

4.4. Comparison with Existing Tools

There exists a variety of tools for static source code checking. In this section, some of them are shortly reviewed and compared with the proposed code checking system. In general, commercial static analysis tools focus on specific domains for analysis. They perform analysis on specific programming languages and some of them focus the analysis on specific purposes such as security or layout. They usually have a set of predefined analysis rules and some of them allow the customization of these rules.

On the other hand, the proposed flexible code checking system is a generic frame-

work and allows the user to build custom checker s for any target language for any domain. The basic advantage of this approach is the ability to adapt and generalize to different domains as opposed to the specific domains of commercial checkers. On the other hand, the advantage of focusing on specific domains is to have a deeper and easier analysis for specific purposes since the main focus of the checker is on those items.

The first tool utilizing source code analysis was *lint*, that flagged suspicious and non-portable constructs (likely to be bugs) in C language source code. An extension of lint called as *Splint*, short for Secure Programming Lint, is a programming tool for statically checking C programs for security vulnerabilities and coding mistakes. Formerly called as *LCLint*, it is a modern version of the lint tool.

One of the commercial tools for static analysis of C++ code is *C++test* of Parasoft [22]. It is a static analyzer for C++ that can be used from command line or can be integrated into leading development environments (IDEs) such as Eclipse or Visual Studio .NET. It contains a number of predefined checks for the C++ language covering many standard analysis rules. It also contains a graphical interface called *RuleWizard* for creating custom checker rules. The graphical wizard allows addition of custom rules and provides flexibility in the analysis. New rules are attached to the rules of the grammar similar to the proposed code checker system but they are designed via a graphical interface. Nevertheless, adding custom rules using this interface is not simple and requires significant level of knowledge and experience. Building a custom rule using *RuleWizard* is depicted in figure 4.1.

The C++Test tool works reliably and is able to produce graphical tool reports on the analysis. There also exists similar tools from Parasoft for Java and .NET languages called as JTest and .TEST respectively.

Programming Research's *QA C++* is another powerful static analysis tool targeting the C++ programming language. It includes lots of predefined analysis rules and the user selects the rules to use in the checker. The tool allows usage of headers and libraries and takes them as command line arguments. It can also be integrated

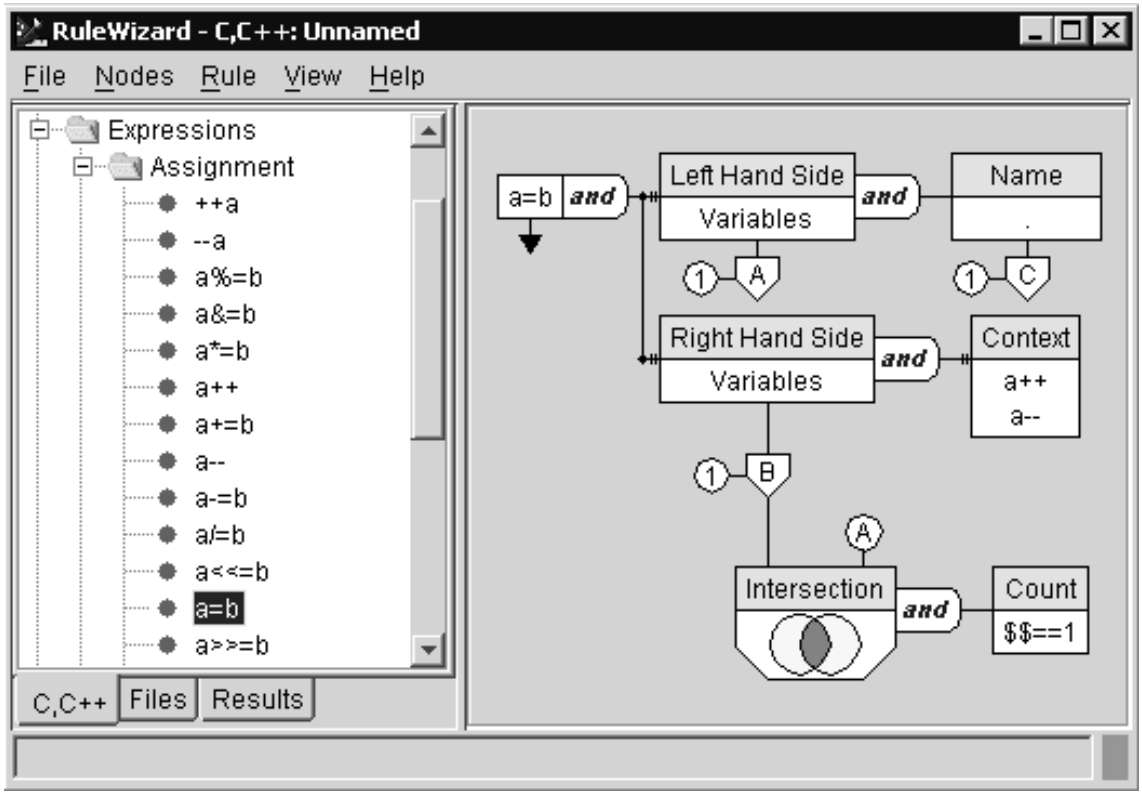

Figure 4.1. Parasoft's C++Test rule wizard

into known IDE's such as Visual Studio .NET. Various forms of quality reports can be obtained from the tool. The tool can also be used to calculate various metrics for the quality of the source code.

5. CONCLUSIONS

In this thesis, a flexible language independent code checker framework is designed. Using the framework, one can build custom checkers for target programming languages by defining the grammar of the target language and by defining the rules of the checker to be applied in the checking of a given source code. The built custom checker is than can be used to analyze source codes in the target programming language and identifying violations of the defined checker rules. Hence, the rules of the checker are completely defined by the user providing the flexibility of the checker.

The main contribution of the proposed code checking framework is providing a flexible and language independent mechanism for building user defined custom checkers as opposed to domain specific code analyzers. Commercial code checkers are designed for helping software developers to analyze source codes in a specific domain for specific purposes for specific target languages. Some of them allow the user to define custom checker rules in that domain.

Instead of concentrating on specific domains and languages, the proposed code checker framework allows the user to build custom checkers in any target language. The user defines the grammar of the target language and specifies the rules for parsing a given source code in that language. The user also defines the checker rules of the checker specifying the conditions to check in the analysis. Different checker rule bases can be defined and used with a single grammar, which allows building checkers for different coding standards on a specific language. Therefore, a checker is completely defined by the user in this framework.

The framework requires programming and target language grammar from the user in defining a checker. However, since the target users are software developers, this does not constitute a major disadvantage. Moreover, the flexibility of the checker comes from this fact. The basic assumption in using the checker is that a given source code for analysis is compilable. Hence, the analysis should be performed after compilation.

Defining the grammar of a target language is not a straightforward task in the framework; in fact it is the most difficult part of building a checker since grammars of programming languages are generally complex. However, once the grammar of a target language is defined, the user can define different checker rule bases for the target grammar.

Two sample checkers, C# checker and C++ checker, are implemented using the proposed flexible code checker framework to demonstrate the flexibility of the checker. The proposed code checker system can be used by software developers for analyzing their codes by their custom rules and finding out the violations of these rules. The results of the check, the list of violations of the checker rules based on their user-defined importance, present the quality of the source code according to the rules of the checker.

The system can also be used in software projects for increasing code quality and maintainability of the source codes in the project, or by instructors for assessing student programs. Furthermore, the framework can also be applied to analyzing any type of text that has a certain representation and can be represented as a grammar in GDML form, thus the framework is not limited to only source codes theoretically.

The performance and the accuracy of the checker depend on the user-defined methods in the grammar and checker definitions respectively. The complexity of the target grammar and the way it is defined using GDML affect the performance of the parsing phase. The amount of recursion in grammar rule items, the complexity of the replacement and checker methods, and even the order of the rule items in grammar rules affect the performance.

The proposed framework can be extended by improving the logical layer for representing different types of constructs in some languages. The framework can be applied to different classes of languages other than object oriented languages, such as markup languages like HTML. Furthermore, code generation capability can be added to the logical layer for generating source codes using the represented constructs similar to that of CodeDOM's but with more expression power. In this way, translation of analyzed

source codes into other languages will be possible.

The framework can also be extended for helping users in defining grammars and checker rules with sample building blocks and visual aids in the user interface without affecting the flexibility of the checker. Furthermore, support for analyzing multiple source files at once in a specified order as a single project can be added to provide deeper analysis in a project.

APPENDIX A: CODE CHECKER CLASS DIAGRAMS

The design process of the Flexible Code Checker can be documented using the class diagrams. This chapter includes the class diagrams of the classes that reside in the Flexible Code Checker.

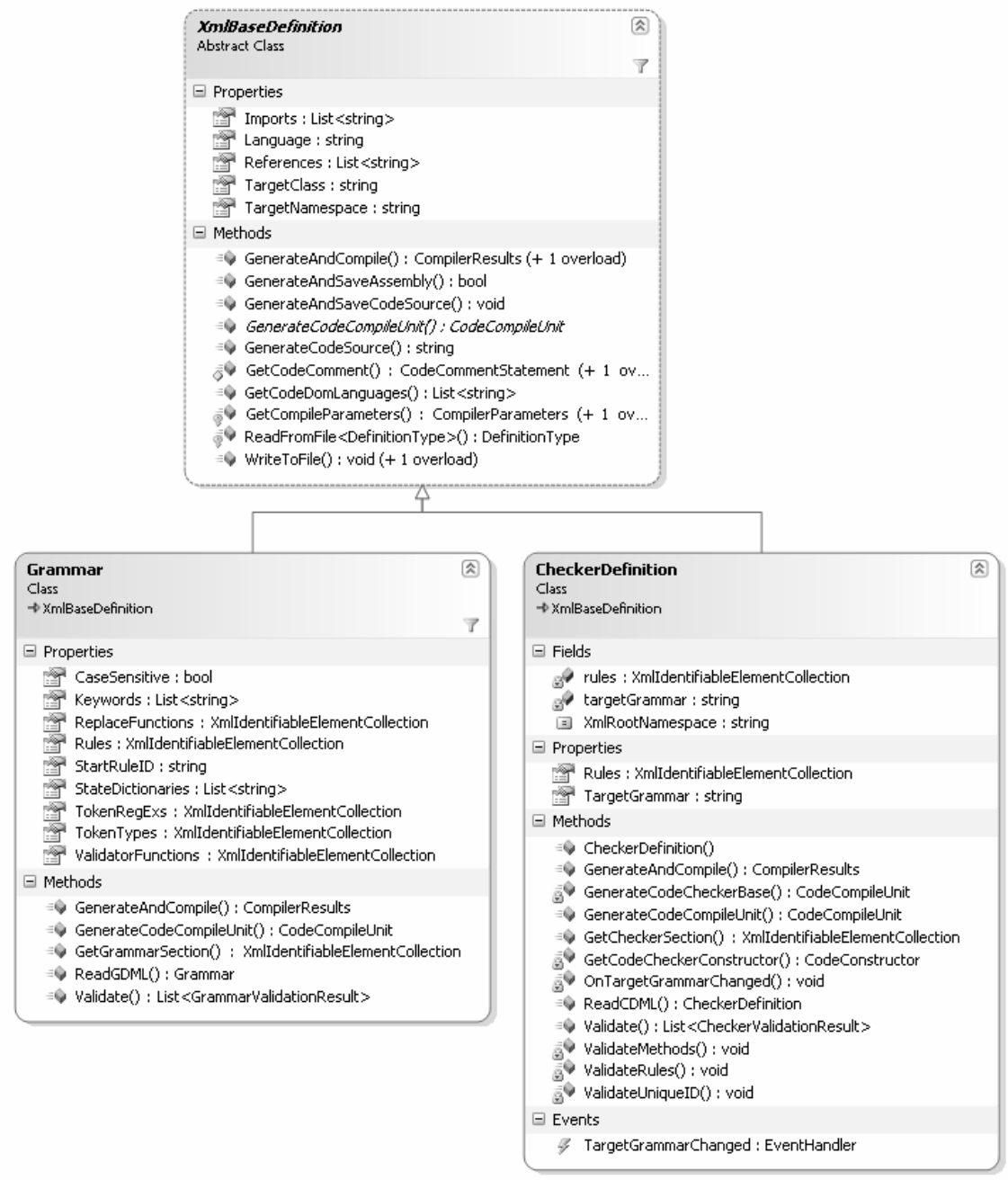

Figure A.1. Code Checker class diagrams - I

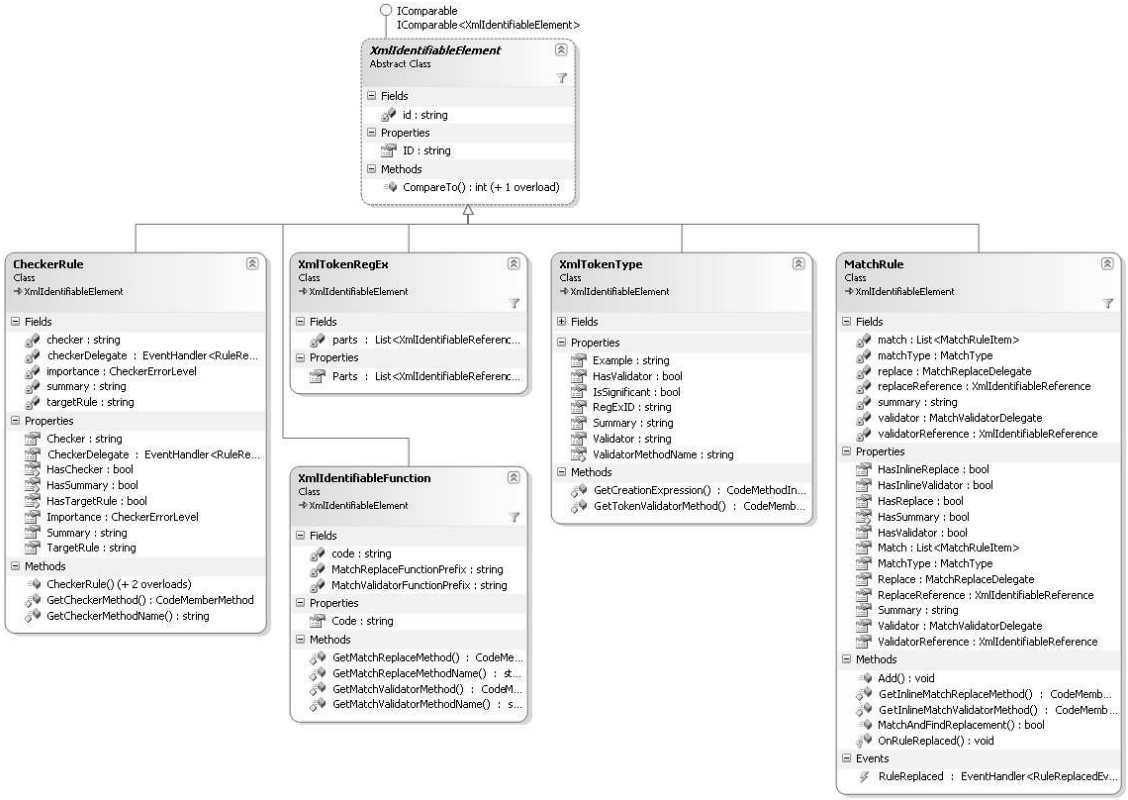

Figure A.2. Code Checker class diagrams - II

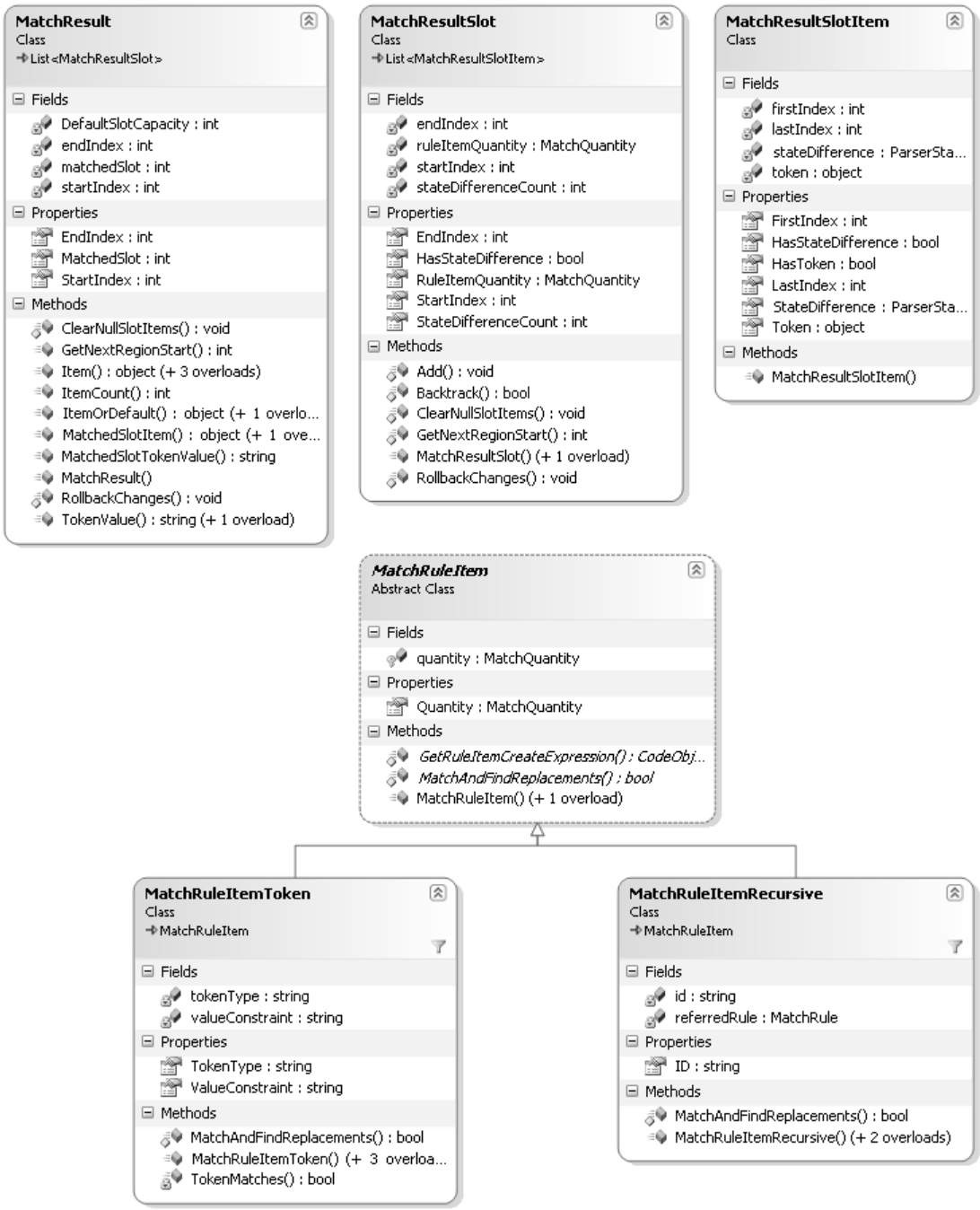

Figure A.3. Code Checker class diagrams - III

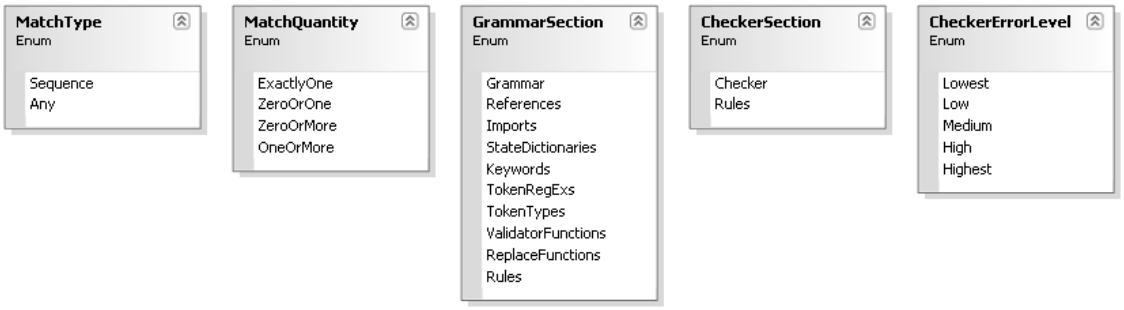

Figure A.4. Code Checker class diagrams - IV

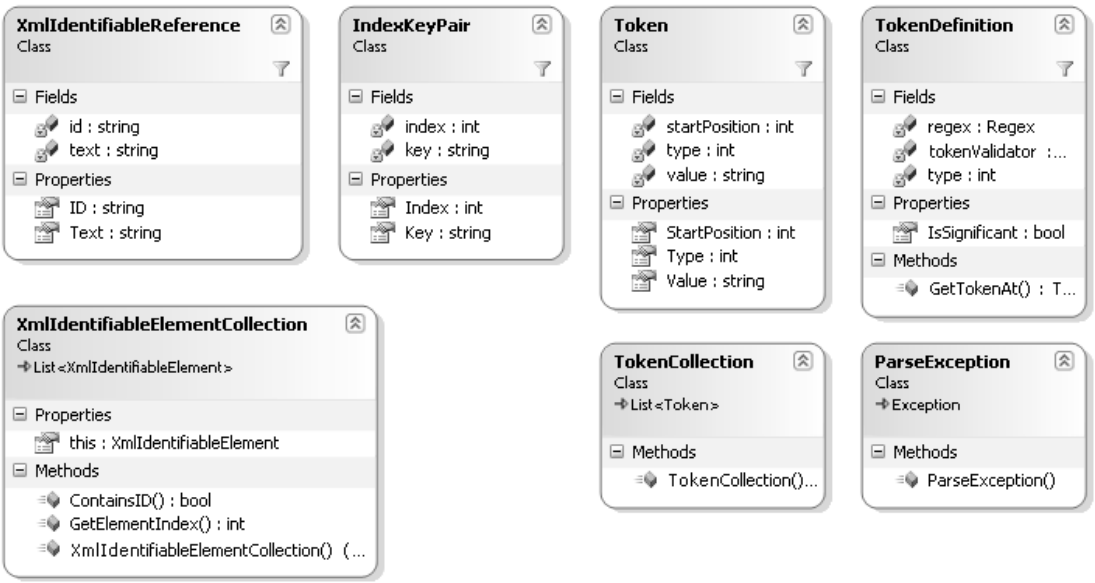

Figure A.5. Code Checker class diagrams - V

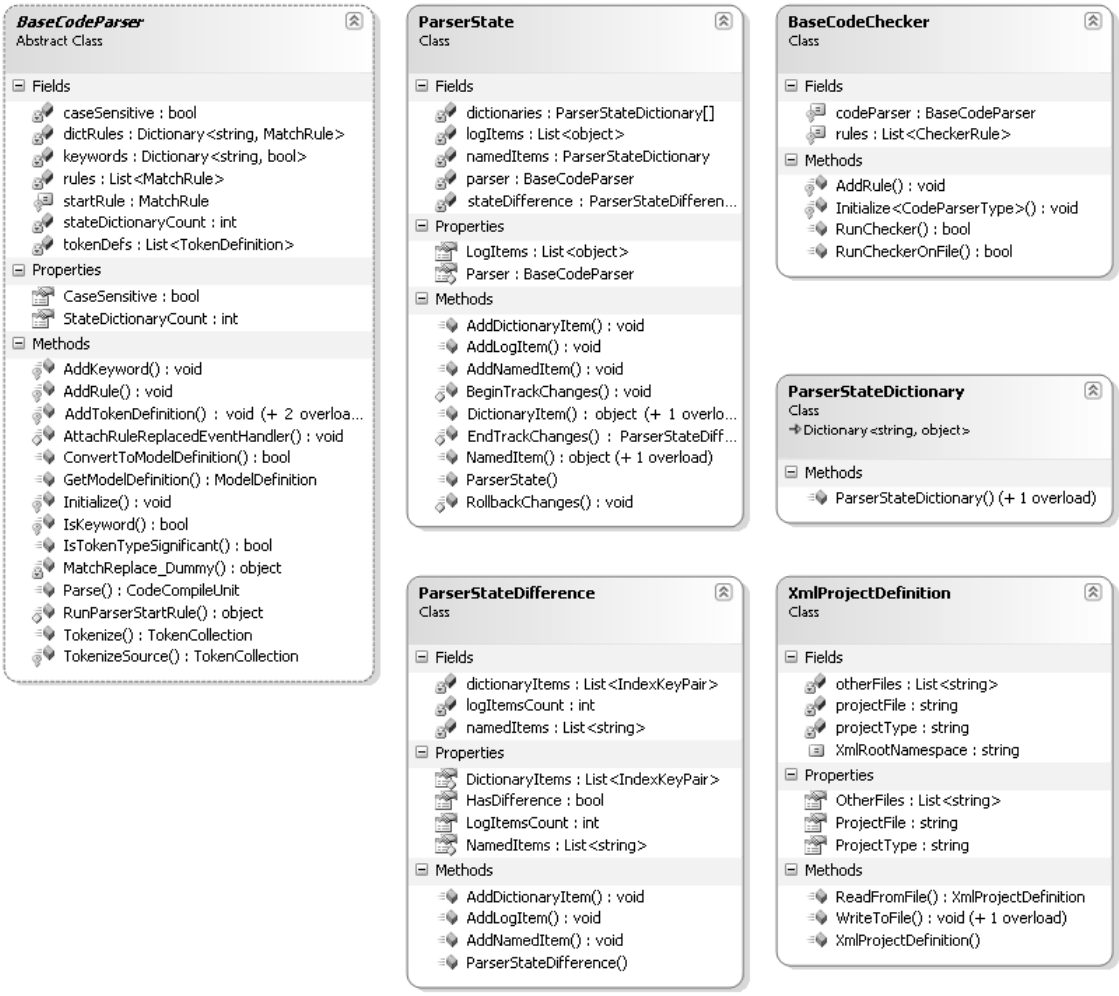

Figure A.6. Code Checker class diagrams - VI

Figure A.7. Code Checker class diagrams - VII

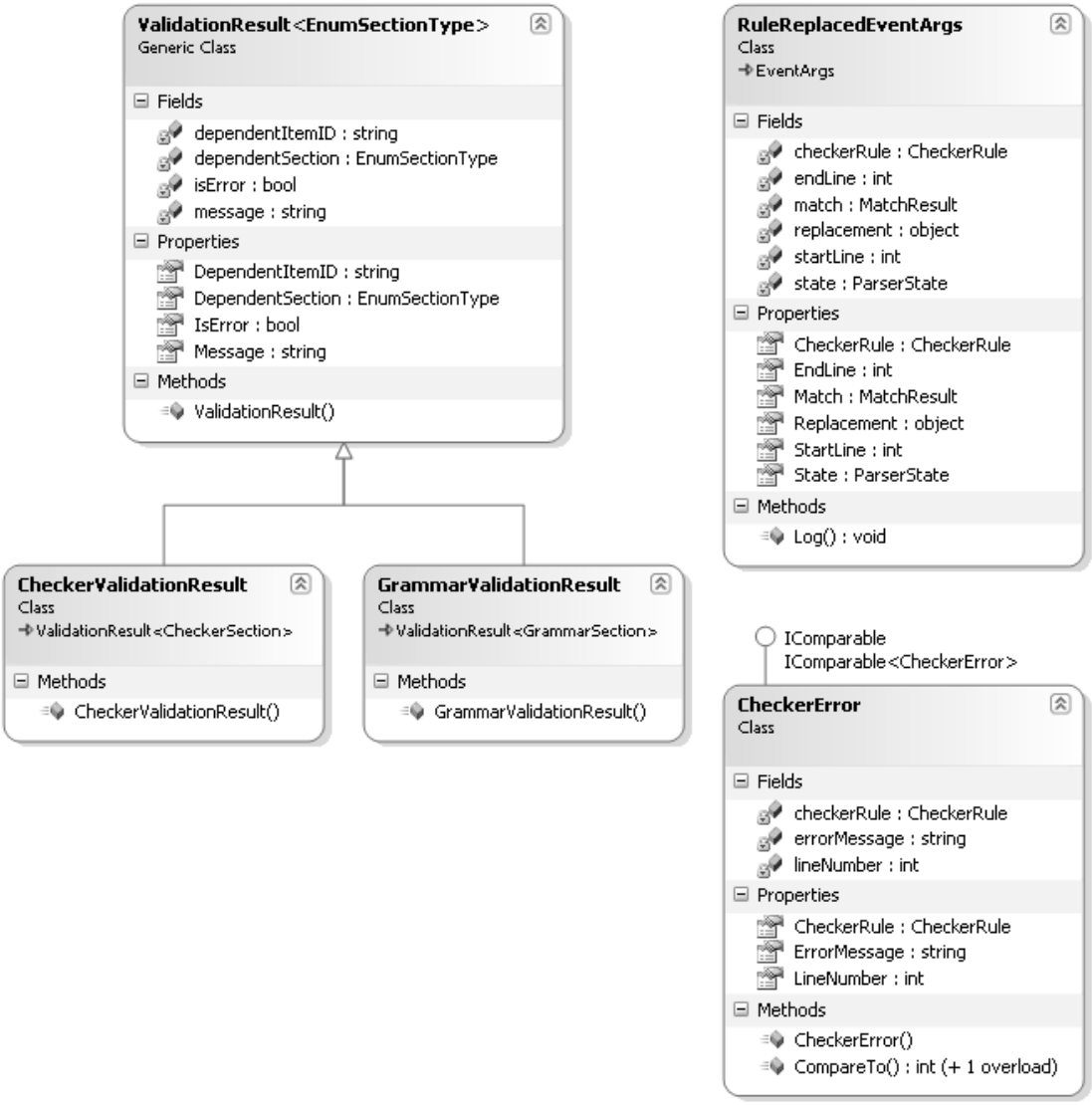

Figure A.8. Code Checker class diagrams - VIII

APPENDIX B: CHECKER LOGICAL LAYER CLASS DIAGRAMS

The class diagrams of the classes that reside in the Logical Layer of the Code Checker are given in this chapter.

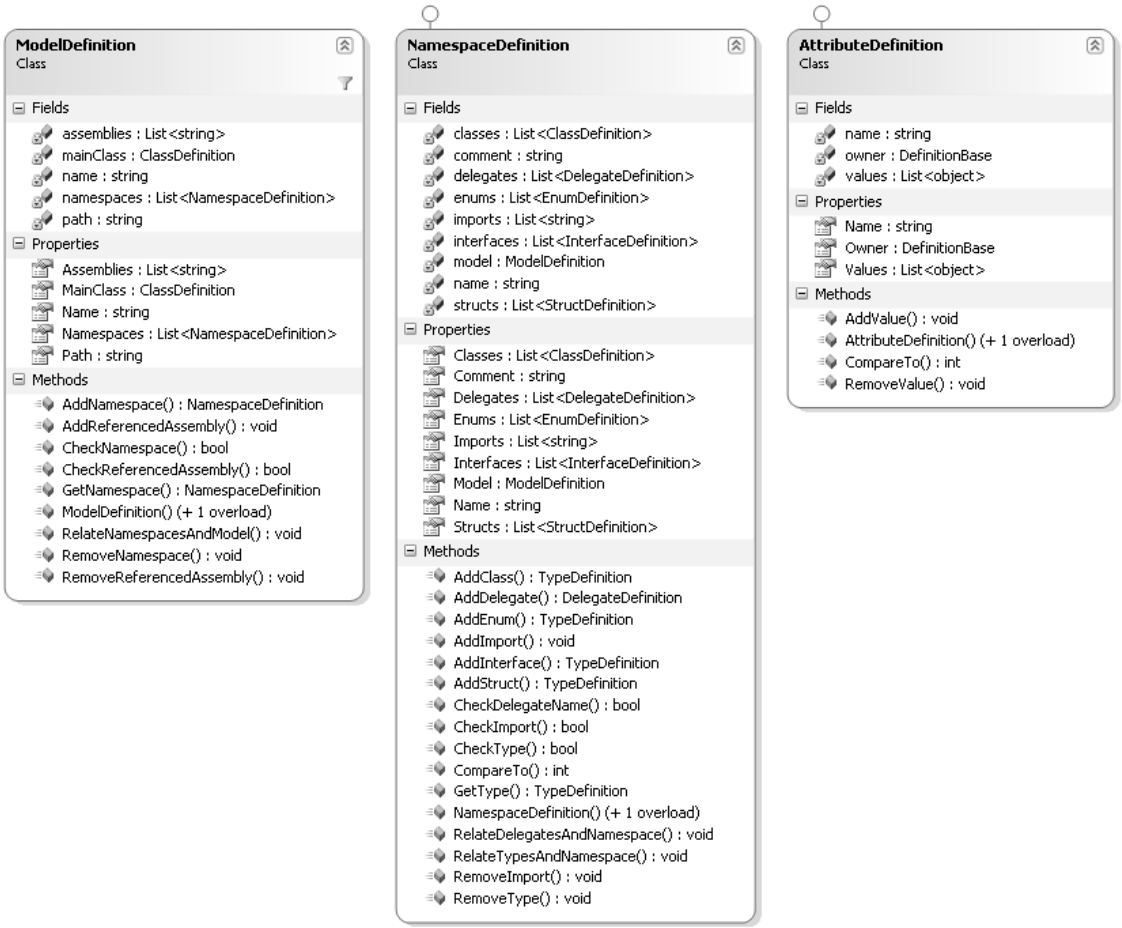

Figure B.1. Code Checker logical layer class diagrams - I

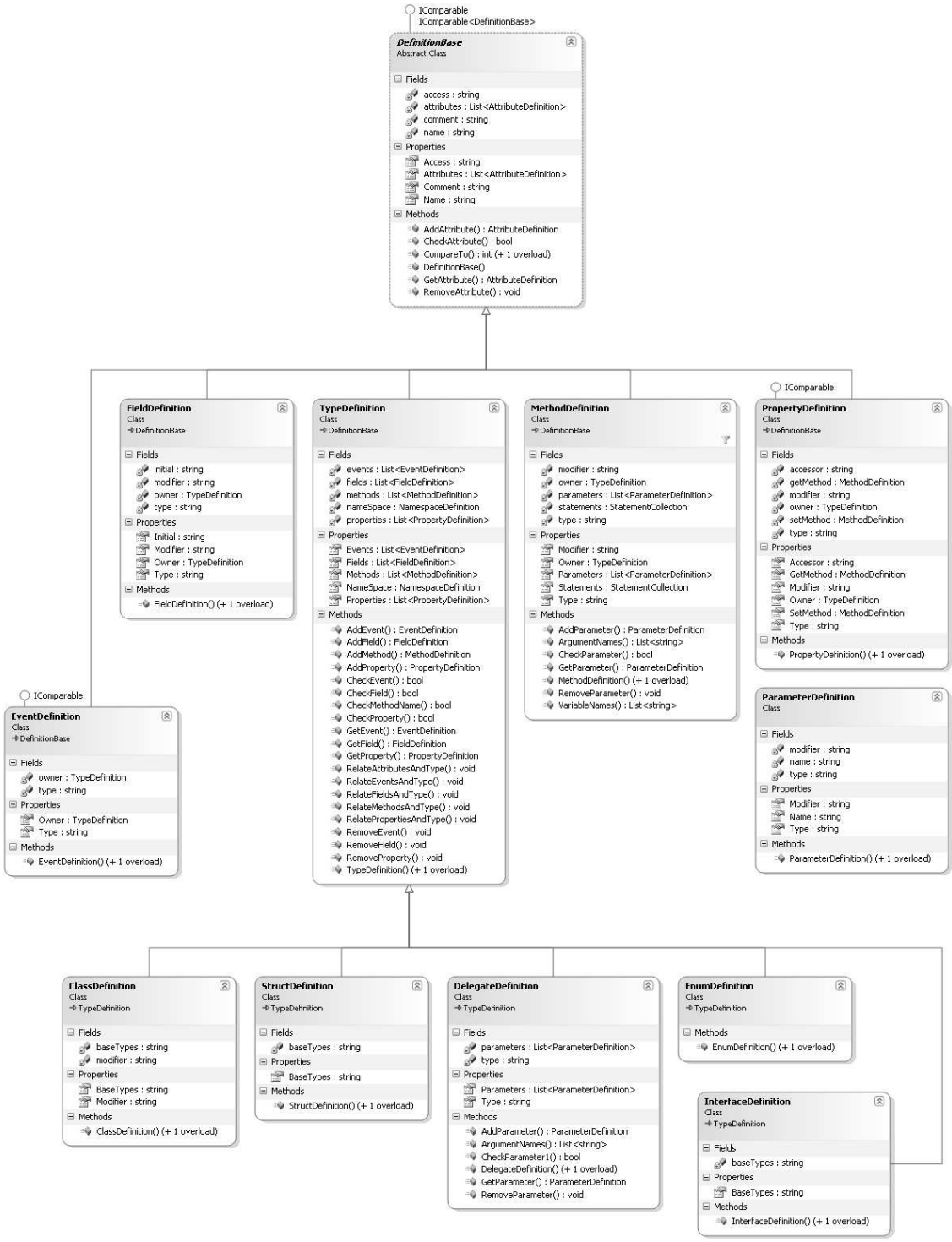

Figure B.2. Code Checker logical layer class diagrams - II

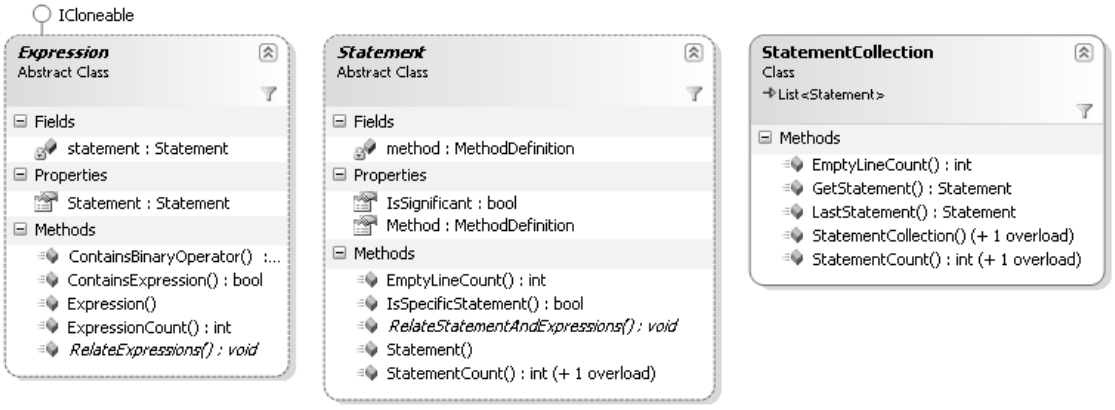

Figure B.3. Code Checker logical layer class diagrams - III

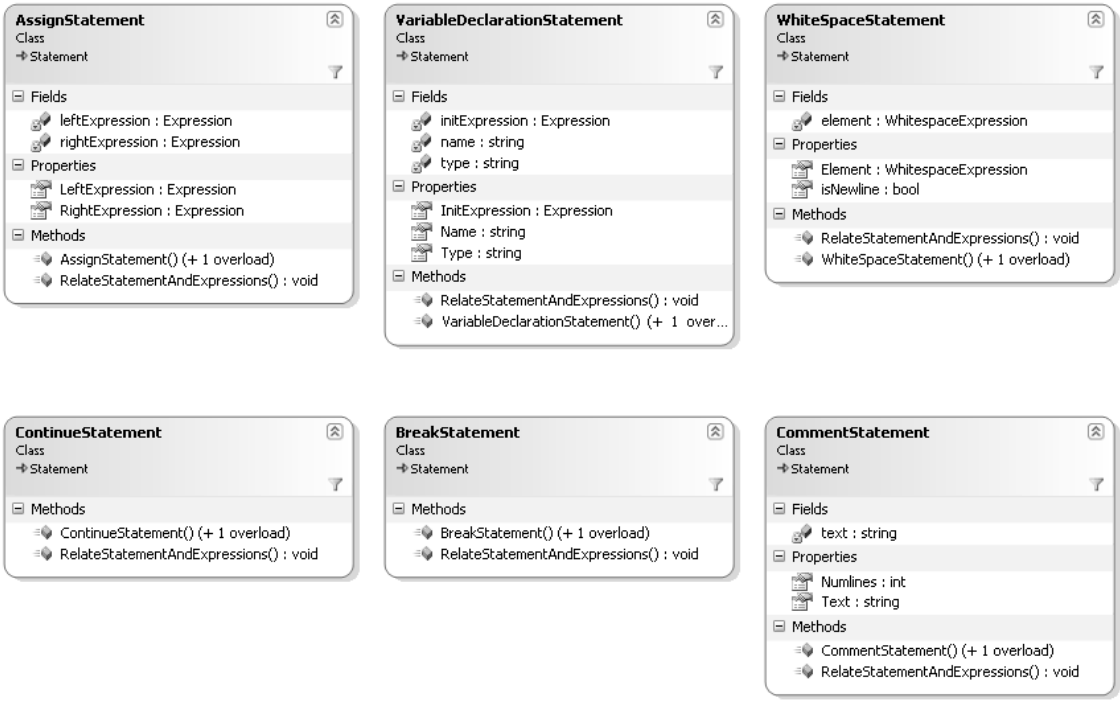

Figure B.4. Code Checker logical layer class diagrams - IV

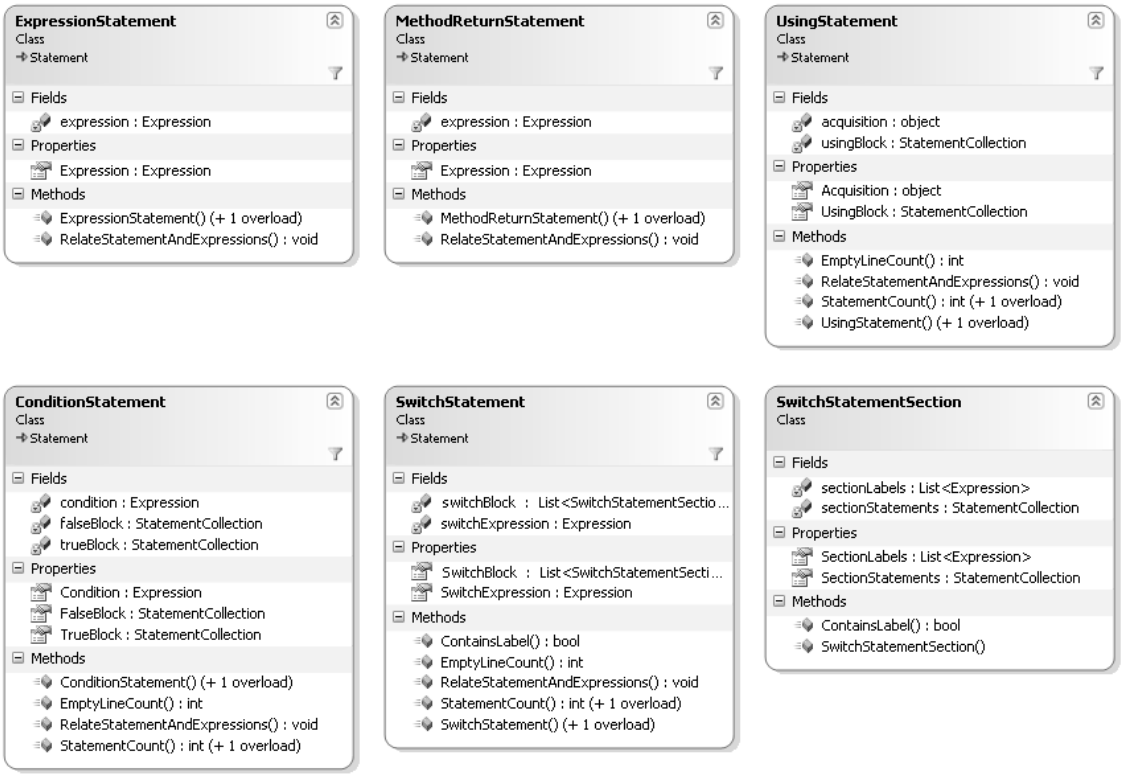

Figure B.5. Code Checker logical layer class diagrams - V

Figure B.6. Code Checker logical layer class diagrams - VI

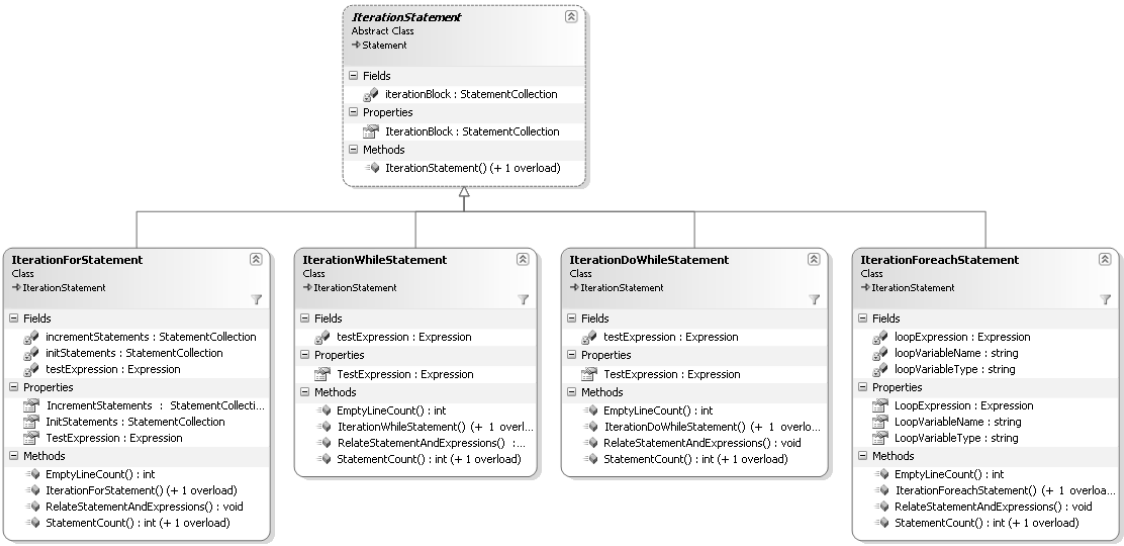

Figure B.7. Code Checker logical layer class diagrams - VII

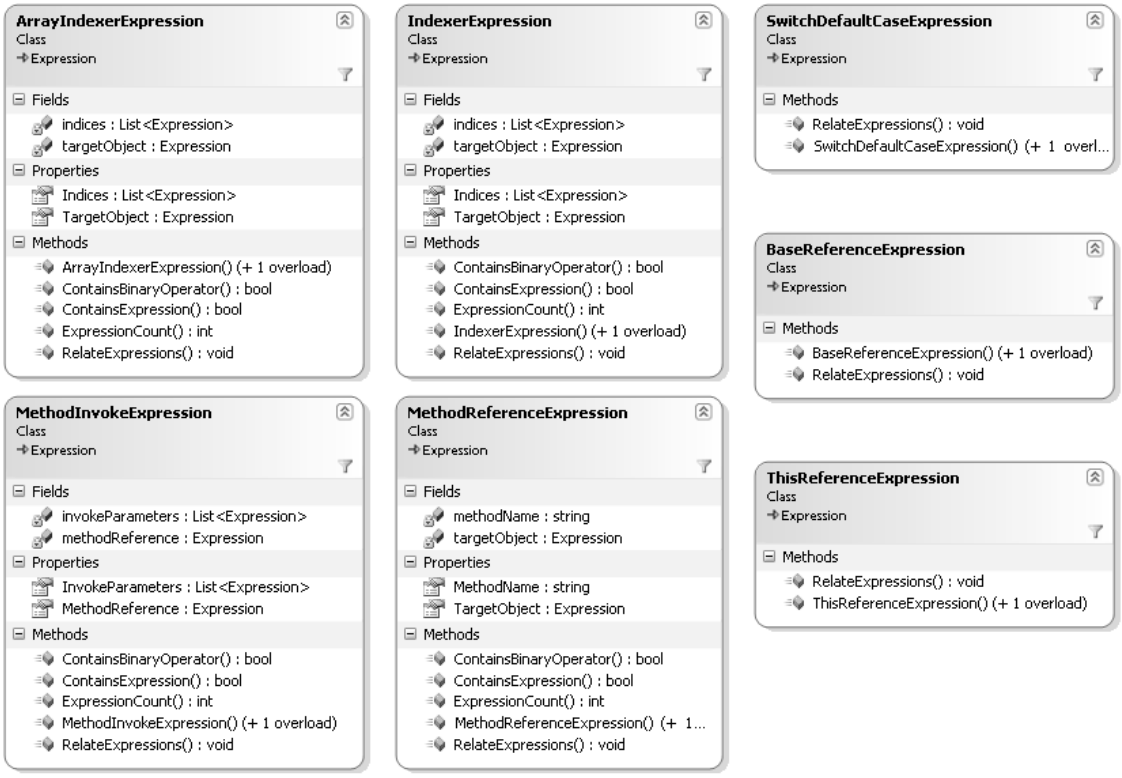

Figure B.8. Code Checker logical layer class diagrams - VIII

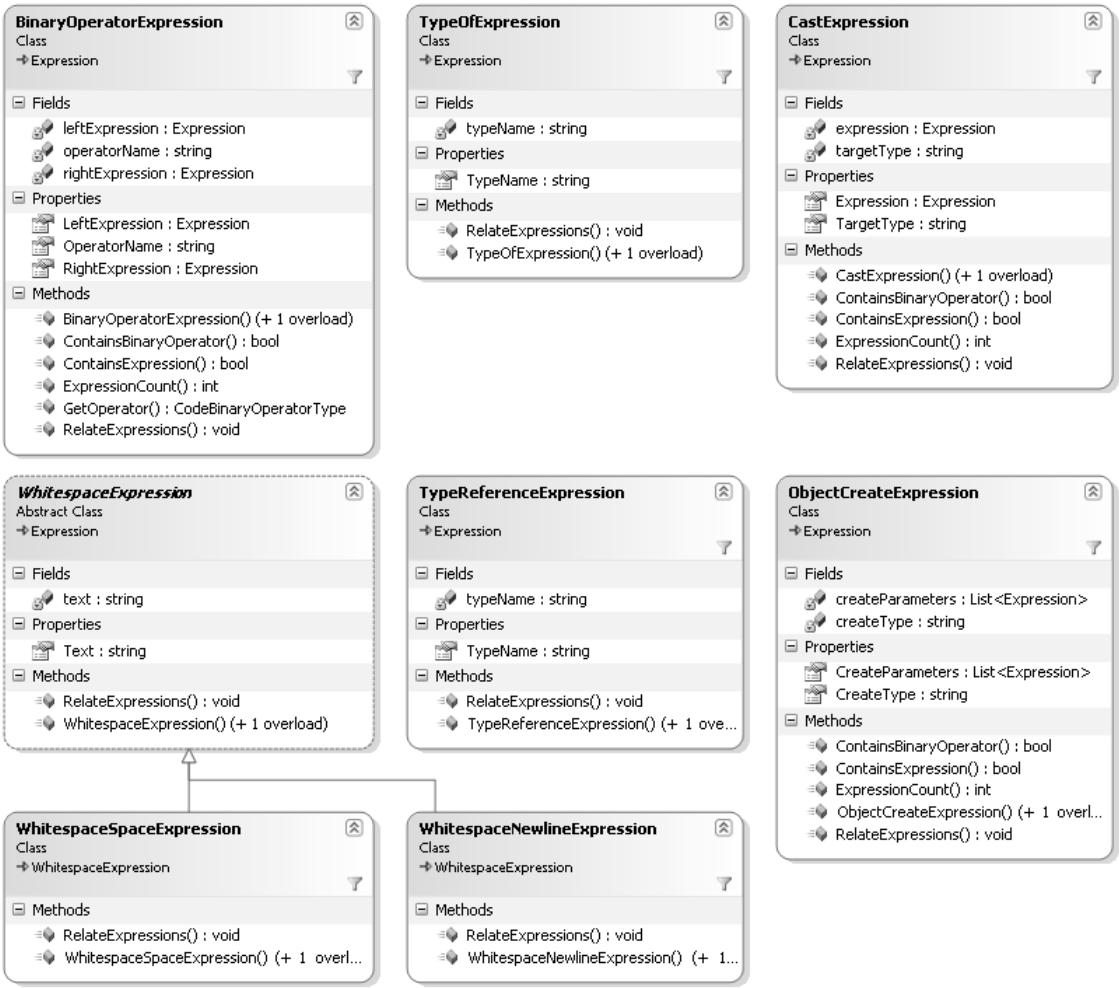

Figure B.9. Code Checker logical layer class diagrams - IX

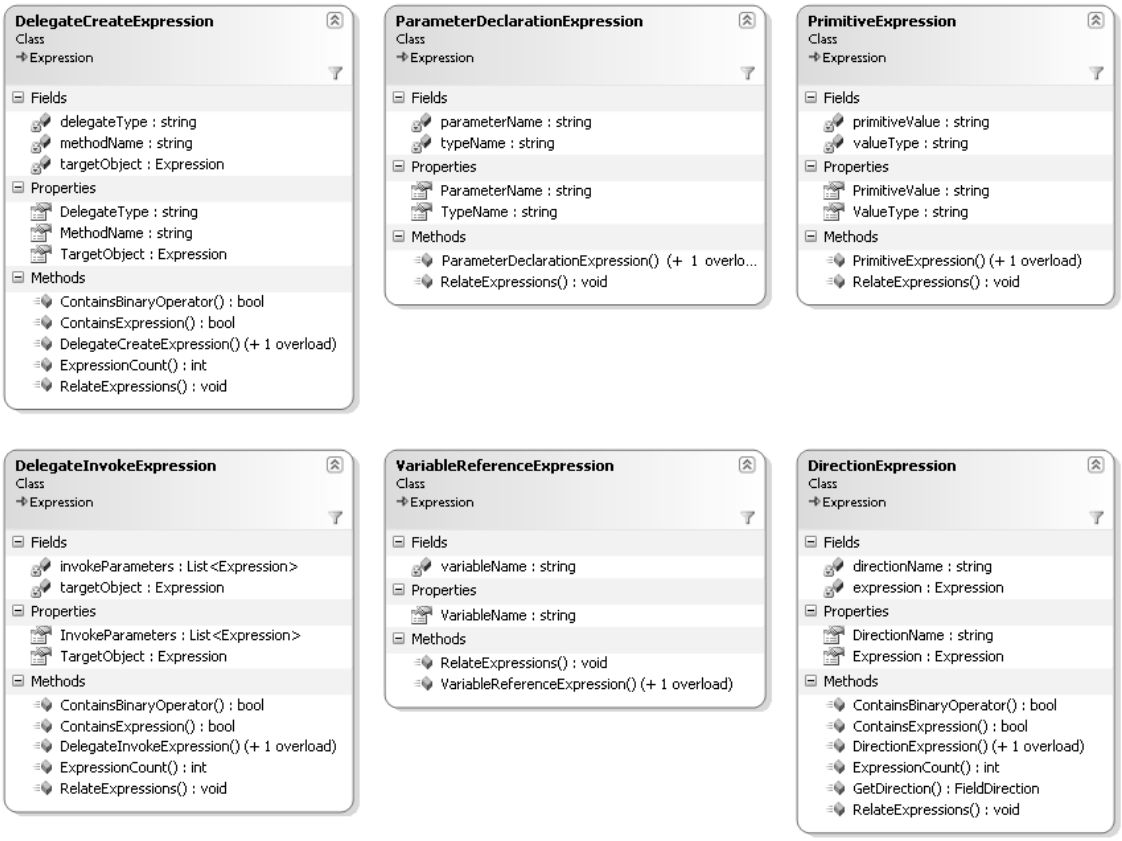

Figure B.10. Code Checker logical layer class diagrams - X

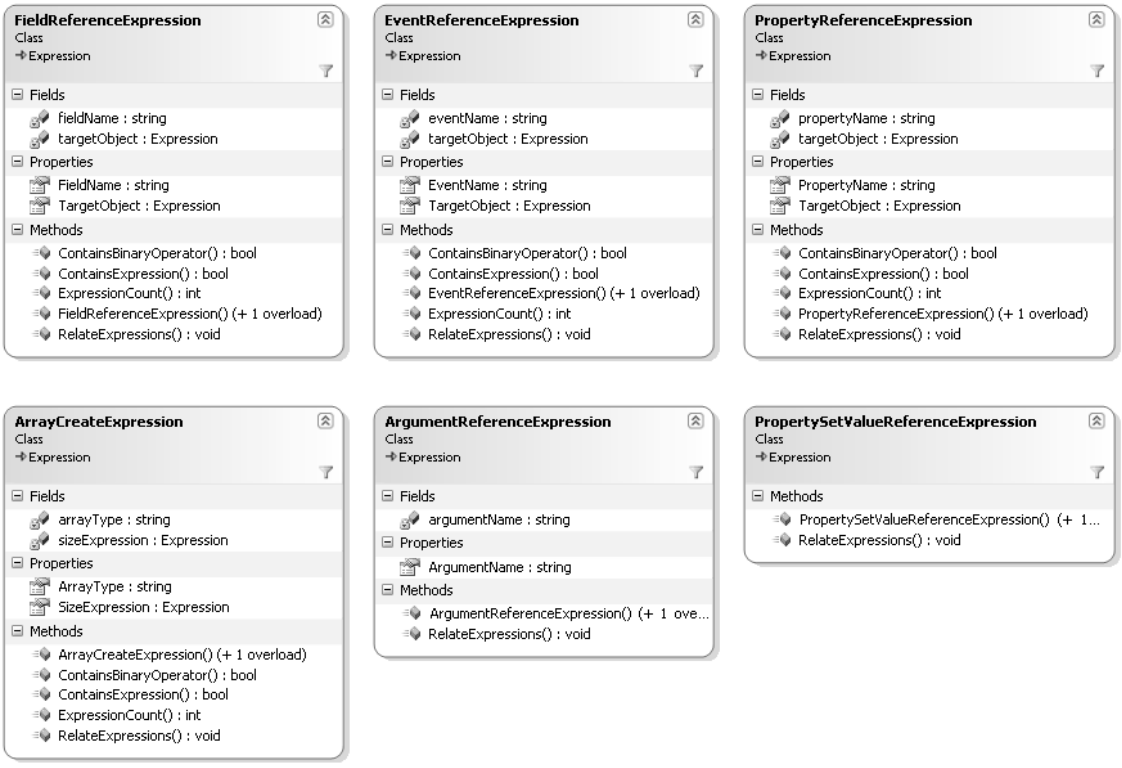

Figure B.11. Code Checker logical layer class diagrams - XI

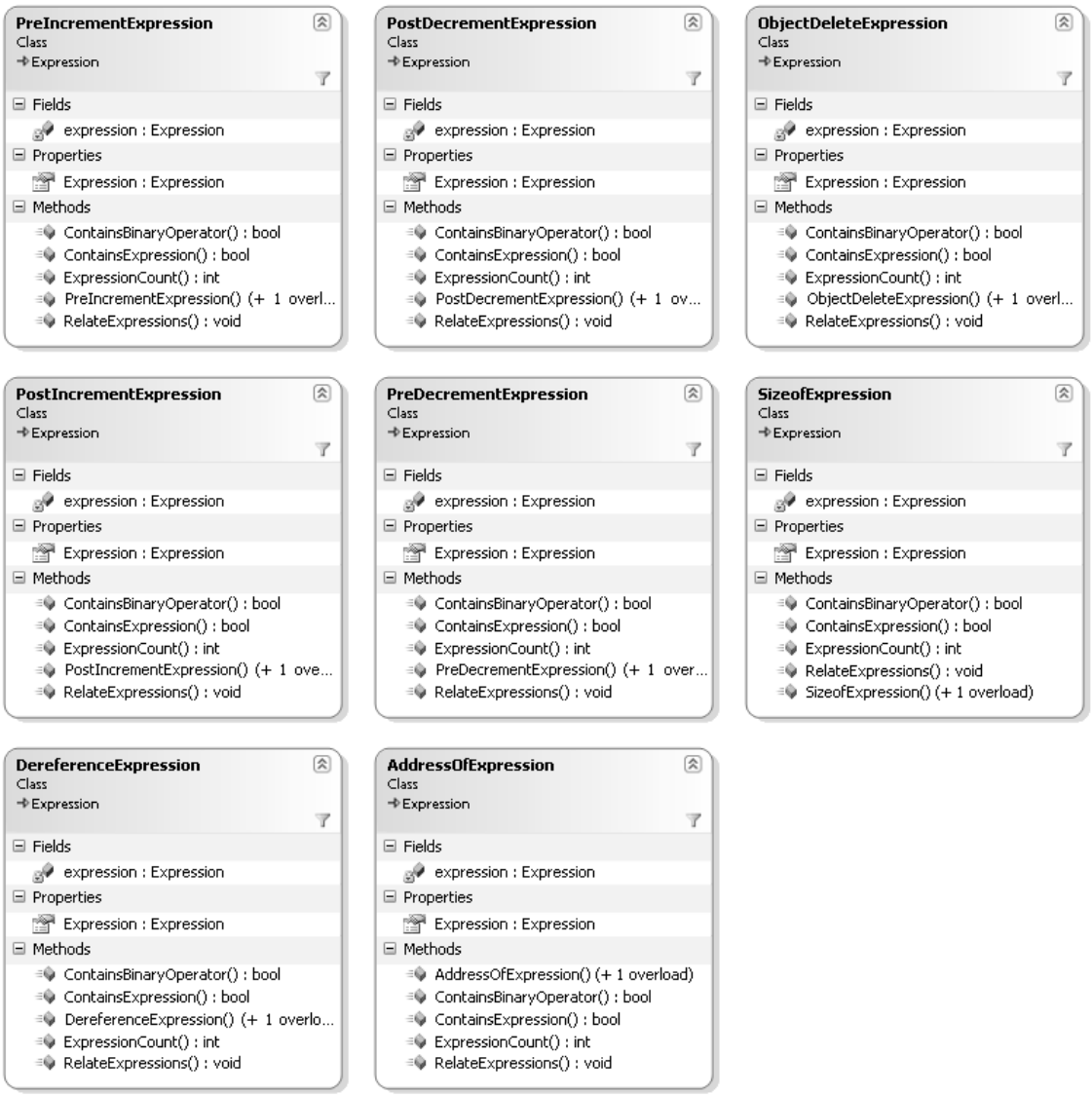

Figure B.12. Code Checker logical layer class diagrams - XII

APPENDIX C: SCREENSHOTS OF FLEXIBLE CODE CHECKER

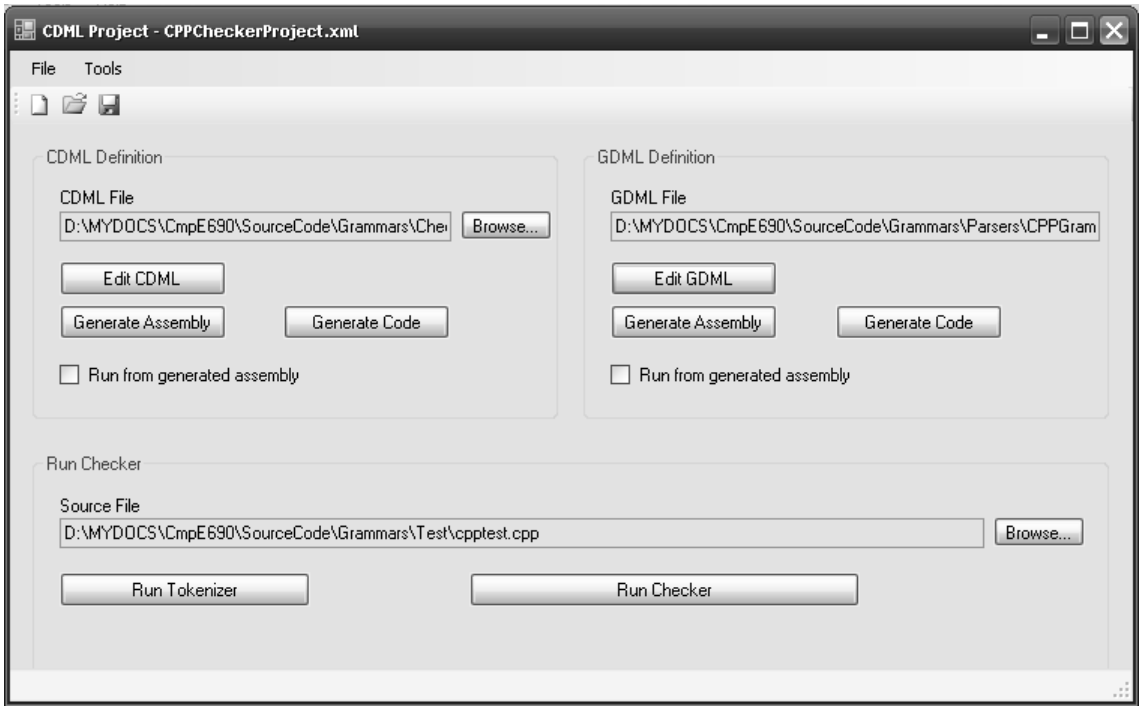

Figure C.1. Code Checker main screen

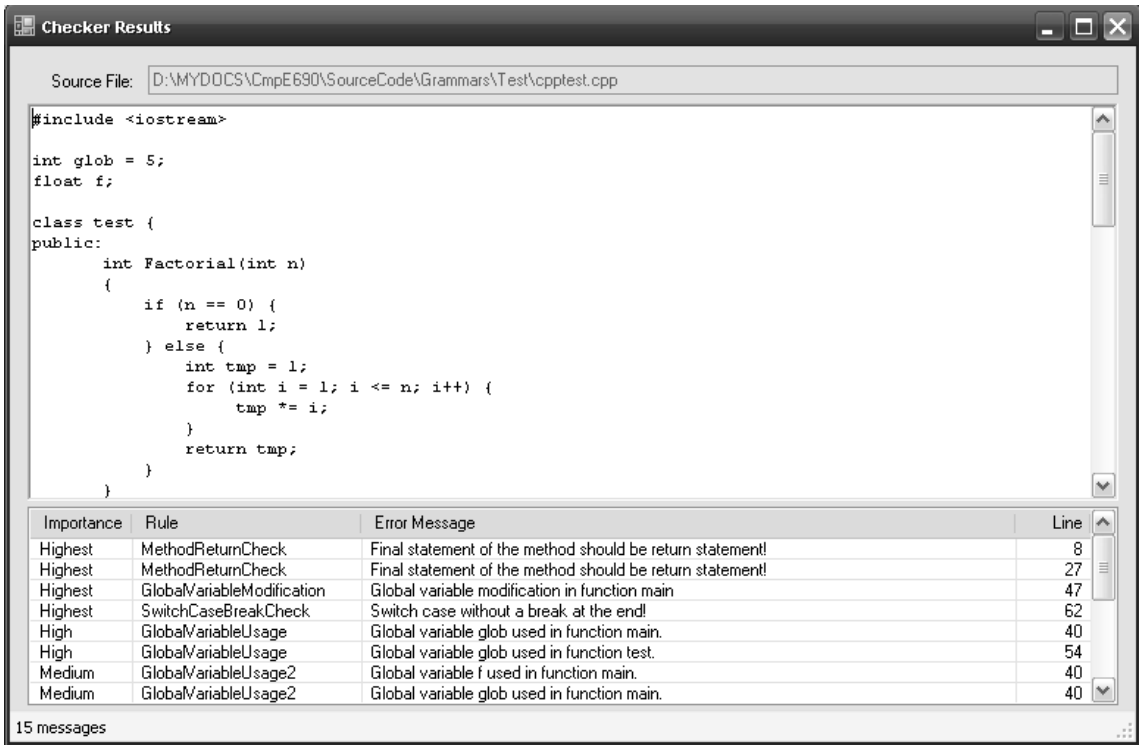

Figure C.2. Code Checker result screen

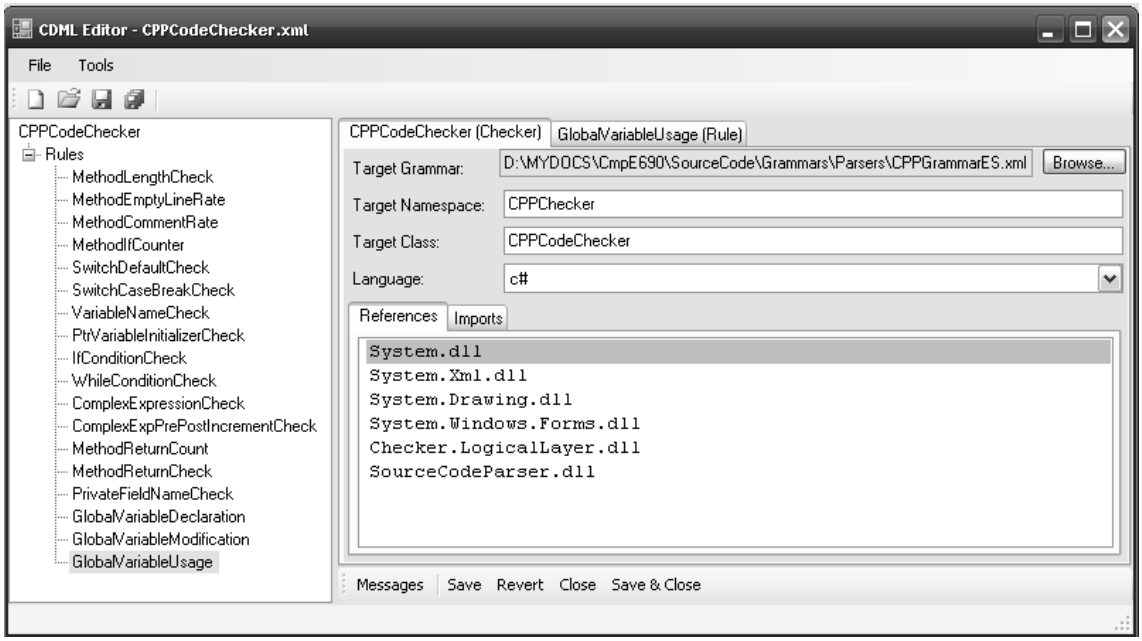

Figure C.3. CDML editor main screen

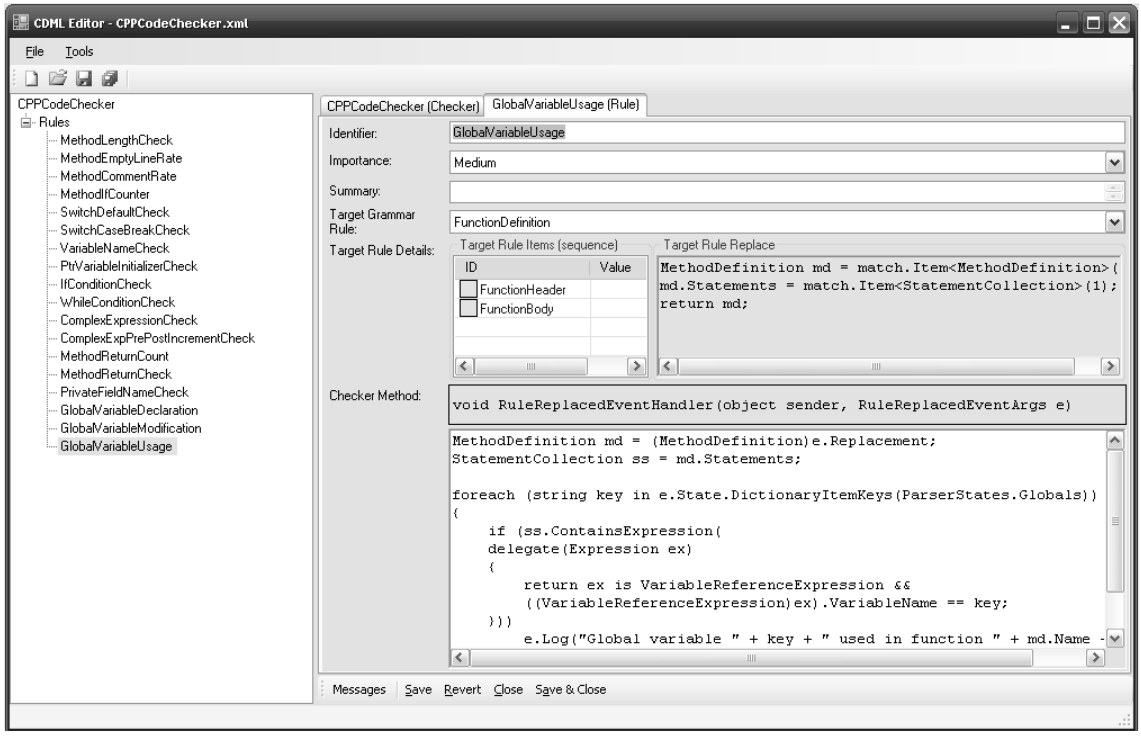

Figure C.4. CDML editor rule edit screen

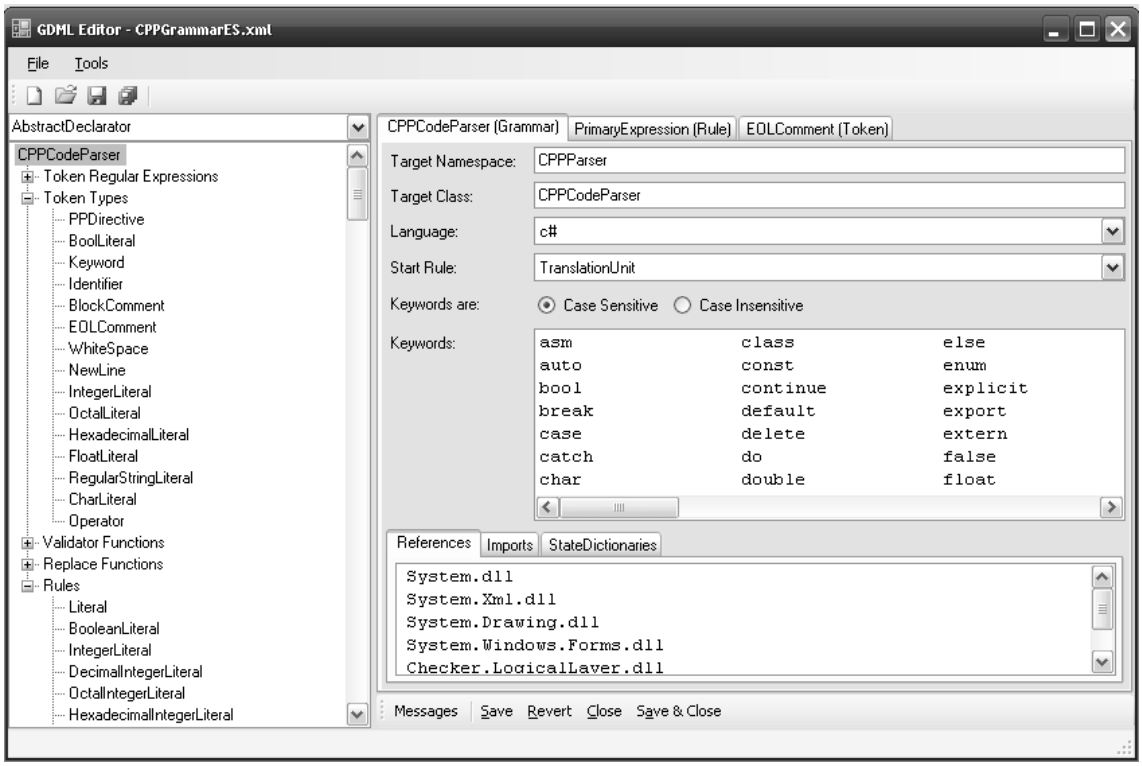

Figure C.5. GDML editor main screen

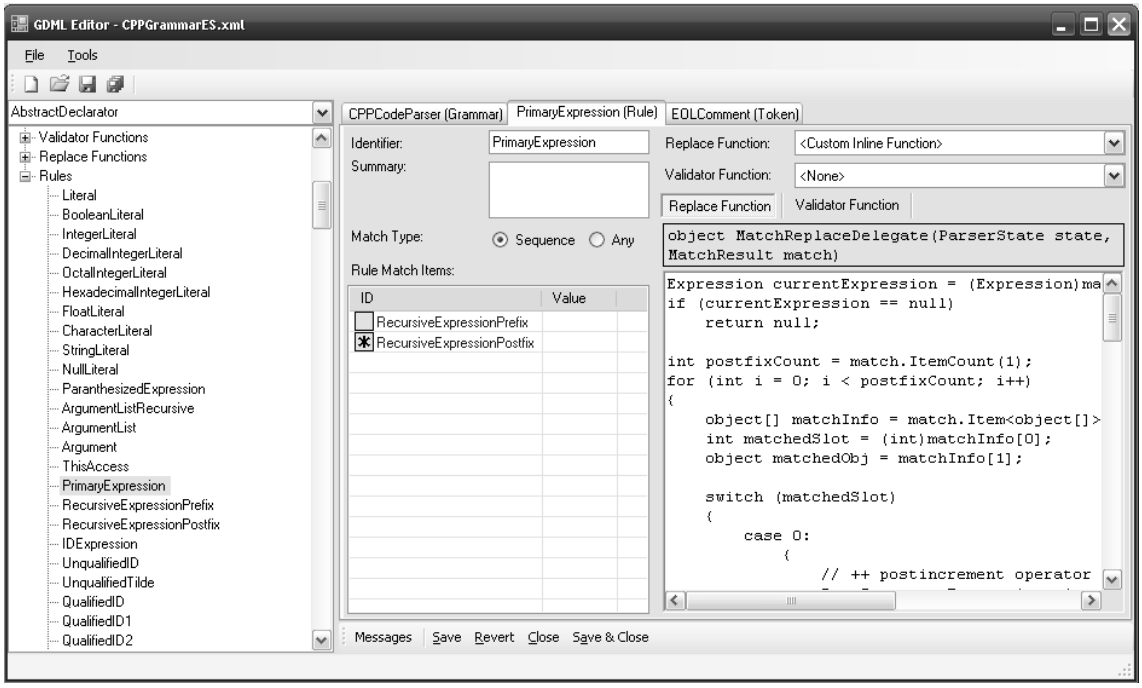

Figure C.6. GDML editor rule edit screen

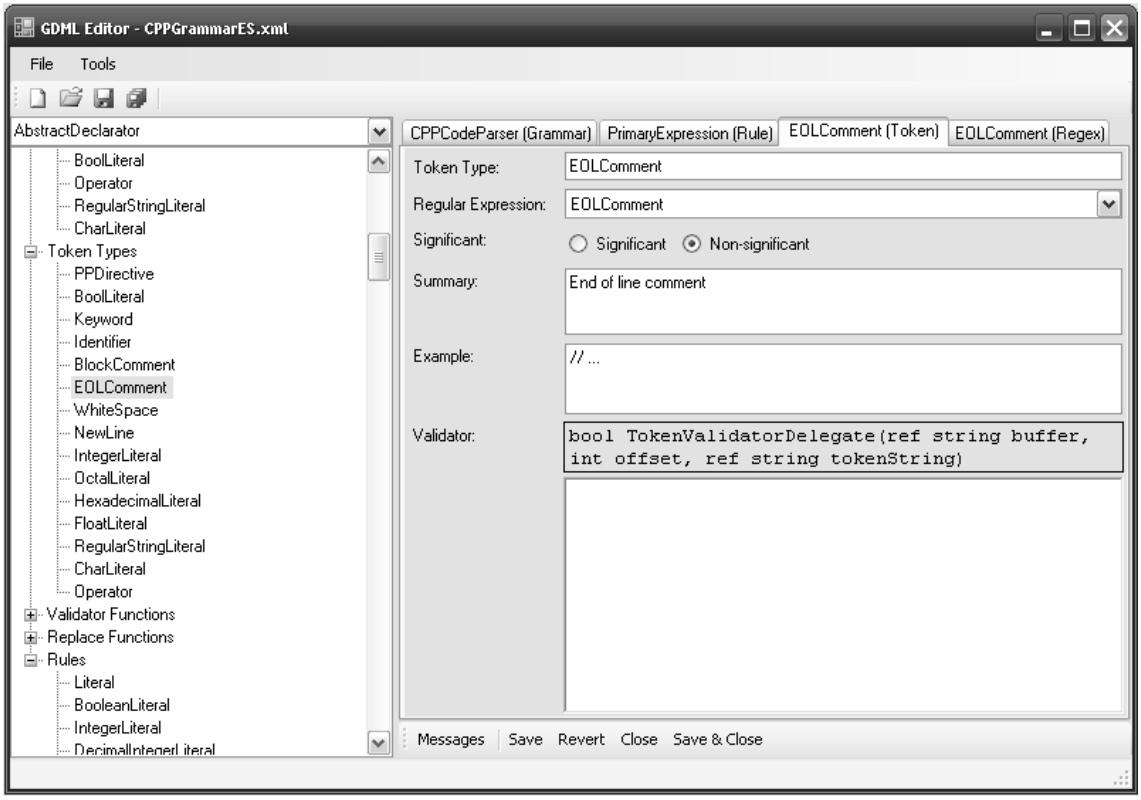

Figure C.7. GDML editor token edit screen

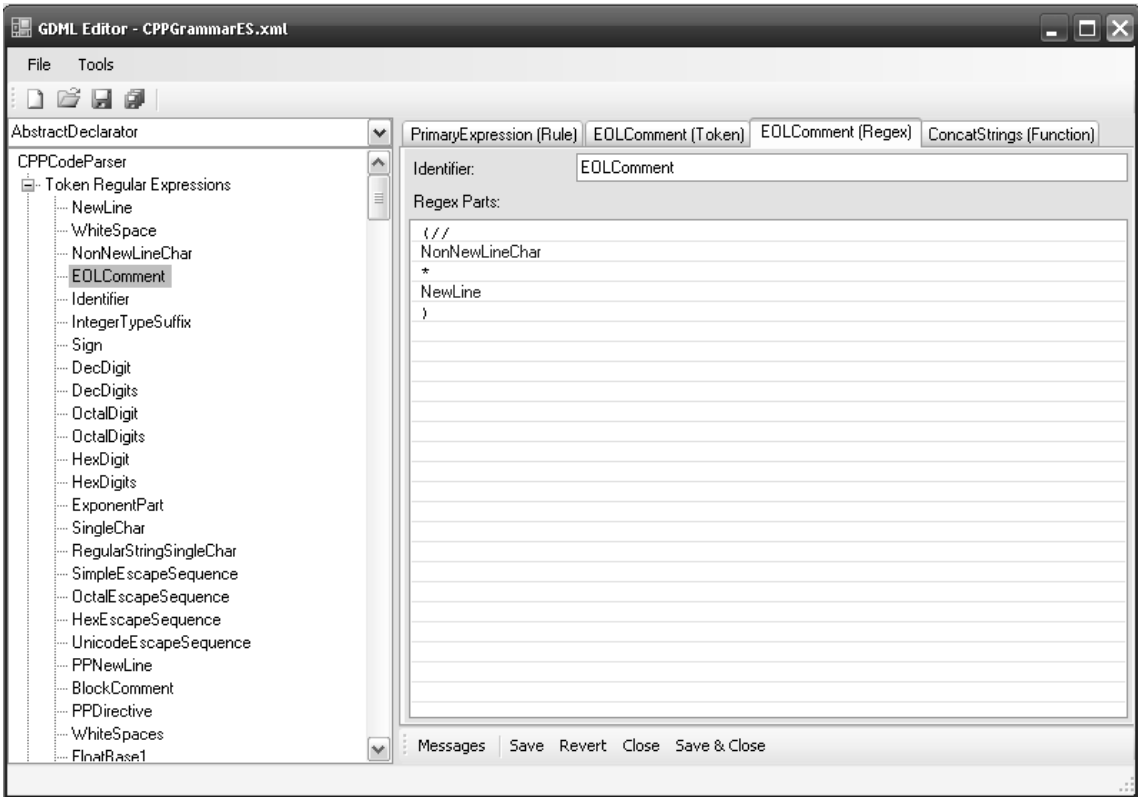

Figure C.8. GDML editor token regex edit screen

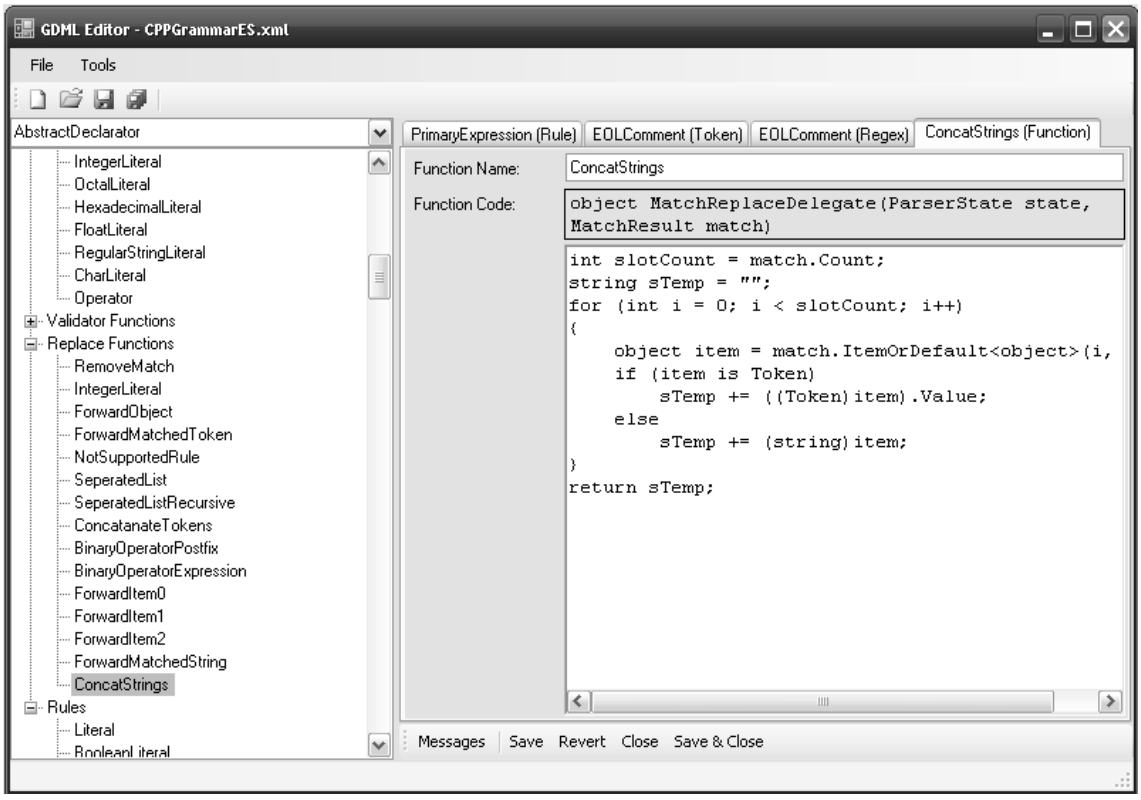

Figure C.9. GDML editor replace function edit screen

APPENDIX D: SAMPLE C++ CHECKER

The details on the sample checker implemented for C++ is given in this chapter. In section D.1, the CDML definition of the checker rules are given in the form of XML. Then in section D.2, the execution of these checker rules on a sample file and the identified violations are given.

D.1. CDML Definition

```
<?xml version="1.0" encoding="utf-8"?>
<Checker xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="CPPChecker" targetClass="CPPCodeChecker"
  targetGrammar="D:\MYDOCS\CmpE690\SourceCode\
    Grammars\Parsers\CPPGrammarES.xml"
  xmlns="http://www.cmpe.boun.edu.tr/checker.xsd">
  <References>
    <Reference>System.dll</Reference>
    <Reference>System.Xml.dll</Reference>
    <Reference>System.Drawing.dll</Reference>
    <Reference>System.Windows.Forms.dll</Reference>
    <Reference>Checker.LogicalLayer.dll</Reference>
    <Reference>SourceCodeParser.dll</Reference>
  </References>
  <Imports>
    <Import>System</Import>
    <Import>System.Collections</Import>
    <Import>System.Collections.Generic</Import>
    <Import>System.ComponentModel</Import>
    <Import>System.Text</Import>
    <Import>System.Windows.Forms</Import>
```

```
  <Import>System.Xml</Import>
  <Import>System.Xml.Serialization</Import>
  <Import>Checker.LogicalLayer</Import>
  <Import>SourceCodeParser</Import>
  <Import>CPPParser</Import>
</Imports>
<Rules>
  <Rule ID="MethodLengthCheck"
    targetRule="FunctionDefinition" importance="low">
    <Summary>Checks the number of effective
      statements of method</Summary>
    <Checker>
      int count =
        e.Match.Item<StatementCollection>(1).StatementCount();
      if (count > 40)
        e.Log("Method statement count too high: " + count);
    </Checker>
  </Rule>
  <Rule ID="MethodEmptyLineRate"
    targetRule="FunctionDefinition" importance="low">
    <Checker>
      int lineCount = e.EndLine - e.StartLine + 1;

      if (lineCount > 50)
        e.Log("Method is too long: "+ lineCount + " lines");

      int emptyLineCount =
        e.Match.Item<StatementCollection>(1).EmptyLineCount();

      int rate = 1000;
      if (emptyLineCount > 0)
        rate = lineCount / emptyLineCount;
```

```
    if (lineCount > 50 && rate > 10)
      e.Log("Method Empty Line/Total Line Rate too low: "
        + emptyLineCount + "/" + lineCount);
    else if (lineCount > 50 && rate < 5)
      e.Log("Method Empty Line/Total Line Rate too high: "
        + emptyLineCount + "/" + lineCount);

  </Checker>
</Rule>
<Rule ID="MethodCommentRate"
  targetRule="FunctionDefinition" importance="lowest">
  <Checker>
    int commentCount =
      e.Match.Item<StatementCollection>(1).StatementCount(
      delegate(Statement s)
      {
        return s is CommentStatement;
      }
    );

    int count =
      e.Match.Item<StatementCollection>(1).StatementCount();
    int rate = 1000;
    if (commentCount > 0)
      rate = count / commentCount;

    if (count > 40 && rate > 10)
      e.Log("Method Comment/Total Statement Rate too low: "
        + commentCount + "/" + count);
  </Checker>
</Rule>
```

```
<Rule ID="MethodIfCounter"
  targetRule="FunctionDefinition" importance="lowest">
  <Summary>Checks the number branching in method.</Summary>
  <Checker>
    int ifCount =
      e.Match.Item<StatementCollection>(1).StatementCount(
      delegate(Statement s)
      {
        return s is ConditionStatement;
      });
    int count =
      e.Match.Item<StatementCollection>(1).StatementCount();
    if (ifCount >= 5)
      e.Log("If/Total Count: " + ifCount + "/" + count);
  </Checker>
</Rule>
<Rule ID="SwitchDefaultCheck"
  targetRule="SwitchStatement" importance="high">
  <Checker>
    SwitchStatement sst = (SwitchStatement)e.Replacement;

    if (!sst.ContainsLabel(
      delegate(Expression exp)
      {
        return exp is SwitchDefaultCaseExpression;
      }))
      e.Log("Switch with no default case");
  </Checker>
</Rule>
<Rule ID="SwitchCaseBreakCheck"
  targetRule="SwitchSection" importance="highest">
  <Checker>
```

```
    StatementCollection ss =
      e.Match.Item<StatementCollection>(1);
    int statementCount = ss.Count;

    if (statementCount > 0 &&
      ss.LastStatement() != null &&
      !ss.LastStatement().IsSpecificStatement(
        delegate(Statement s)
        {
          return s is BreakStatement;
        }))
      e.Log("Switch case without a break at the end!");
  </Checker>
</Rule>
<Rule ID="VariableNameCheck"
  targetRule="SimpleDeclaration" importance="lowest">
  <Summary>
    Checks whether integer variable names start with _.
  </Summary>
  <Checker>
    StatementCollection vds =
      (StatementCollection)e.Replacement;

    foreach (VariableDeclarationStatement vd in vds)
    {
      if (vd.Type == "int" &&
        vd.Name.StartsWith("_"))
        e.Log("Local integer variables" +
          " should not begin with _.");
    }
  </Checker>
</Rule>
```

```
<Rule ID="PtrVariableInitializerCheck"
  targetRule="SimpleDeclaration">
  <Summary>
    Checks whether pointer variables are initialized
    in declaration.
  </Summary>
  <Checker>
    StatementCollection vds =
      (StatementCollection)e.Replacement;

    foreach (VariableDeclarationStatement vd in vds)
    {
      if (vd.Type.Contains("*")
        && vd.InitExpression == null)
        e.Log("Uninitialized pointer variable declaration: "
          + vd.Name);
    }
  </Checker>
</Rule>
<Rule ID="IfConditionCheck"
  targetRule="IfStatement" importance="highest">
  <Summary>Checks for assignment in if condition</Summary>
  <Checker>
    Expression condition = e.Match.Item<Expression>(0);

    if (condition.ContainsBinaryOperator("assign"))
      e.Log("Use of assignment in If condition");
  </Checker>
</Rule>
<Rule ID="WhileConditionCheck"
  targetRule="WhileStatement" importance="highest">
  <Summary>Checks for assignment in while condition</Summary>
```

```
  <Checker>
    Expression condition = e.Match.Item<Expression>(0);

    if (condition.ContainsBinaryOperator("assign"))
      e.Log("Use of assignment in While condition");
  </Checker>
</Rule>
<Rule ID="ComplexExpressionCheck"
  targetRule="ExpressionStatement">
  <Summary>
    Checks expressions for the number of subexpressions
    included
  </Summary>
  <Checker>
    Expression ex = e.Match.Item<Expression>(0);
    int count = ex.ExpressionCount();

    if (count > 10)
      e.Log("Complex expression usage.");
  </Checker>
</Rule>
<Rule ID="ComplexExpPrePostIncrementCheck"
  targetRule="ExpressionStatement" importance="highest">
  <Summary>
    Checks usage of pre/post increment and decrement
    expressions other than stand-alone
  </Summary>
  <Checker>
    Expression ex = e.Match.Item<Expression>(0);

    // pre/post increment-decrement should be stand-alone
    if (ex.ExpressionCount() > 2)
```

```
    {
      if (ex.ContainsExpression(
        delegate(Expression expr)
        {
          return expr is PostIncrementExpression ||
          expr is PostDecrementExpression;
        }))
        e.Log("Post increment/decrement expressions" +
          " should be used stand-alone.");

      if (ex.ContainsExpression(
        delegate(Expression expr)
        {
          return expr is PreIncrementExpression ||
          expr is PreDecrementExpression;
        }))
        e.Log("Pre increment/decrement expressions" +
          " should be used stand-alone.");
    }
  </Checker>
</Rule>
<Rule ID="MethodReturnCount"
  targetRule="FunctionDefinition">
  <Summary>Checks number of return points of a method.</Summary>
  <Checker>
    int returnCount =
      e.Match.Item<StatementCollection>(1).StatementCount(
      delegate(Statement s)
      {
        return s is MethodReturnStatement;
      });
```

```
    if (returnCount > 2)
      e.Log("Too many return points in method");
  </Checker>
</Rule>
<Rule ID="MethodReturnCheck"
  targetRule="FunctionDefinition" importance="highest">
  <Summary>
    Checks whether the final statement of a non-void
    method is return statement
  </Summary>
  <Checker>
    MethodDefinition md = (MethodDefinition)e.Replacement;
    StatementCollection ss = md.Statements;
    int statementCount = ss.Count;

    if (statementCount > 0 &&
      md.Type != "void" &&
      ss.LastStatement() != null &&
      !ss.LastStatement().IsSpecificStatement(
        delegate(Statement s)
        {
          return s is MethodReturnStatement;
        }
      ))
      e.Log("Final statement of the method" +
        " should be return statement!");
  </Checker>
</Rule>
<Rule ID="PrivateFieldNameCheck"
  targetRule="ClassSpecifier" importance="lowest">
  <Summary>
    Checks class fields should be private and
```

```
    the private field names should start with _
  </Summary>
  <Checker>
    ClassDefinition cd = (ClassDefinition)e.Replacement;

    foreach (FieldDefinition fd in cd.Fields)
    {
      if (fd.Access.Contains("private"))
      {
        if (!fd.Name.StartsWith("_"))
          e.Log("Private fields should begin with _.");
      }
      else
        e.Log("Nonprivate field: " + fd.Name + " in class.");
    }

  </Checker>
</Rule>
<Rule ID="GlobalVariableDeclaration"
  targetRule="SimpleDeclaration" importance="low">
  <Summary>Check for a global variable declaration.</Summary>
  <Checker>
    if (e.State.HasNamedItem("CurrentMethod") ||
      e.State.HasNamedItem("CurrentClass"))
      return;

    StatementCollection vds =
      (StatementCollection)e.Replacement;

    foreach (VariableDeclarationStatement vd in vds)
      e.Log("Global variable declaration: " + vd.Name);
  </Checker>
```

```
</Rule>
<Rule ID="GlobalVariableModification"
  targetRule="Assignment" importance="highest">
  <Summary>
    Checks for modification of global variables
    in functions
  </Summary>
  <Checker>
    Expression lhs = e.Match.Item<Expression>(0);

    if (lhs is VariableReferenceExpression &&
      e.State.HasNamedItem("CurrentMethod"))
    {
      MethodDefinition md =
        e.State.NamedItem<MethodDefinition>("CurrentMethod");

      VariableReferenceExpression vre =
        lhs as VariableReferenceExpression;

      if (e.State.HasDictionaryItem(
        ParserStates.Globals, vre.VariableName))
        e.Log("Global variable modification in function "
          + md.Name);
    }
  </Checker>
</Rule>
<Rule ID="GlobalVariableUsage"
  targetRule="FunctionDefinition">
  <Summary>
    Checks the usage of each global variable within function.
  </Summary>
  <Checker>
```

```
        MethodDefinition md = (MethodDefinition)e.Replacement;
        StatementCollection ss = md.Statements;

        foreach (string key in
          e.State.DictionaryItemKeys(ParserStates.Globals))
        {
          if (ss.ContainsExpression(
            delegate(Expression ex)
            {
              return ex is VariableReferenceExpression &&
                ((VariableReferenceExpression)ex).VariableName == key;
            }))
            e.Log("Global variable " + key +
              " used in function " + md.Name + ".");
        }
      </Checker>
    </Rule>
  </Rules>
</Checker>
```

D.2. Checker Results

In this section, the analysis results of the sample C++ checker are given. The results are depicted in the following figures, where in each figure the sample analyzed source file is on the top and the list of violations including the error messages and line numbers are listed below.

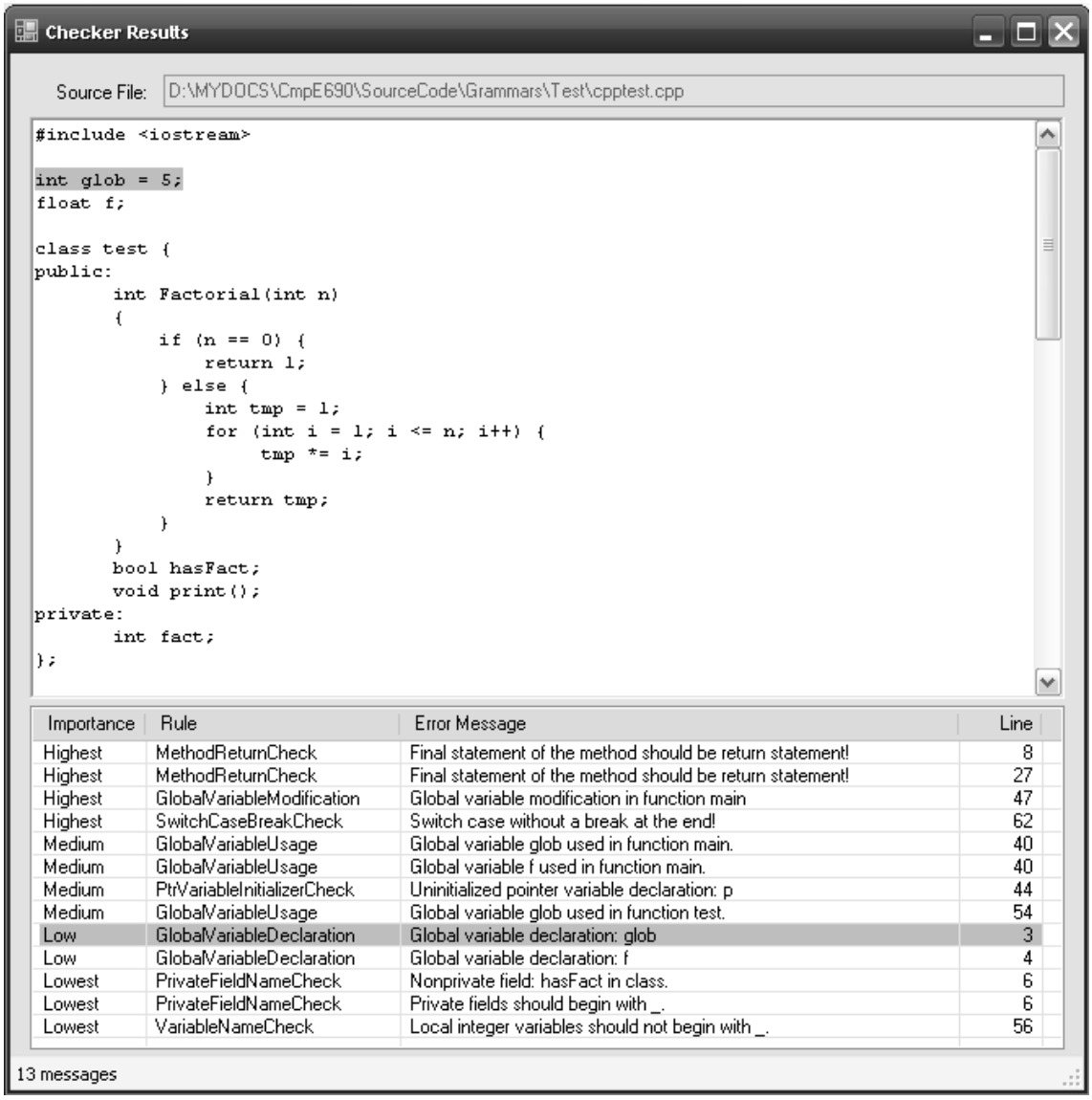

Figure D.1. Sample C++ checker results - I

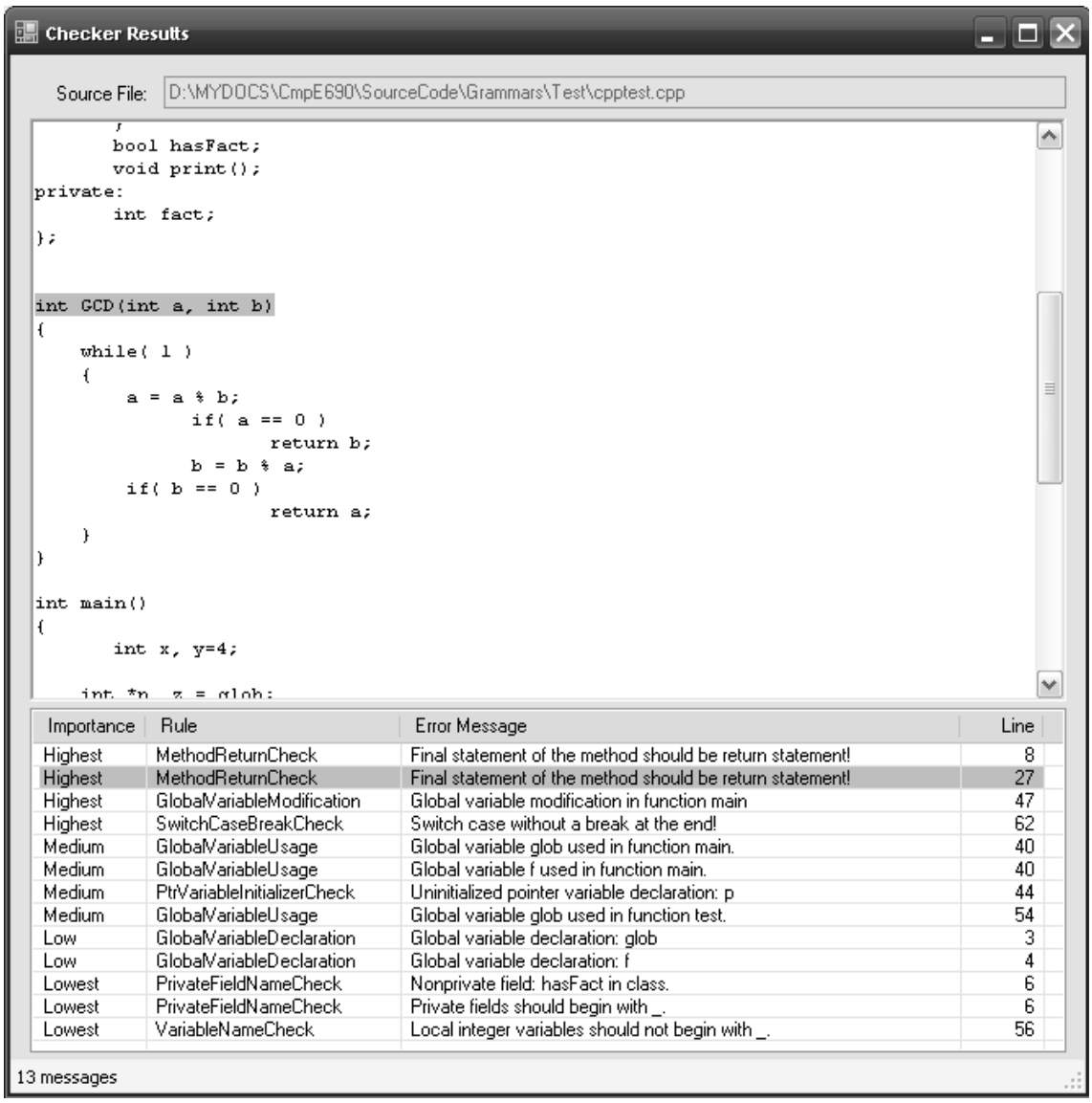

Figure D.2. Sample C++ checker results - II

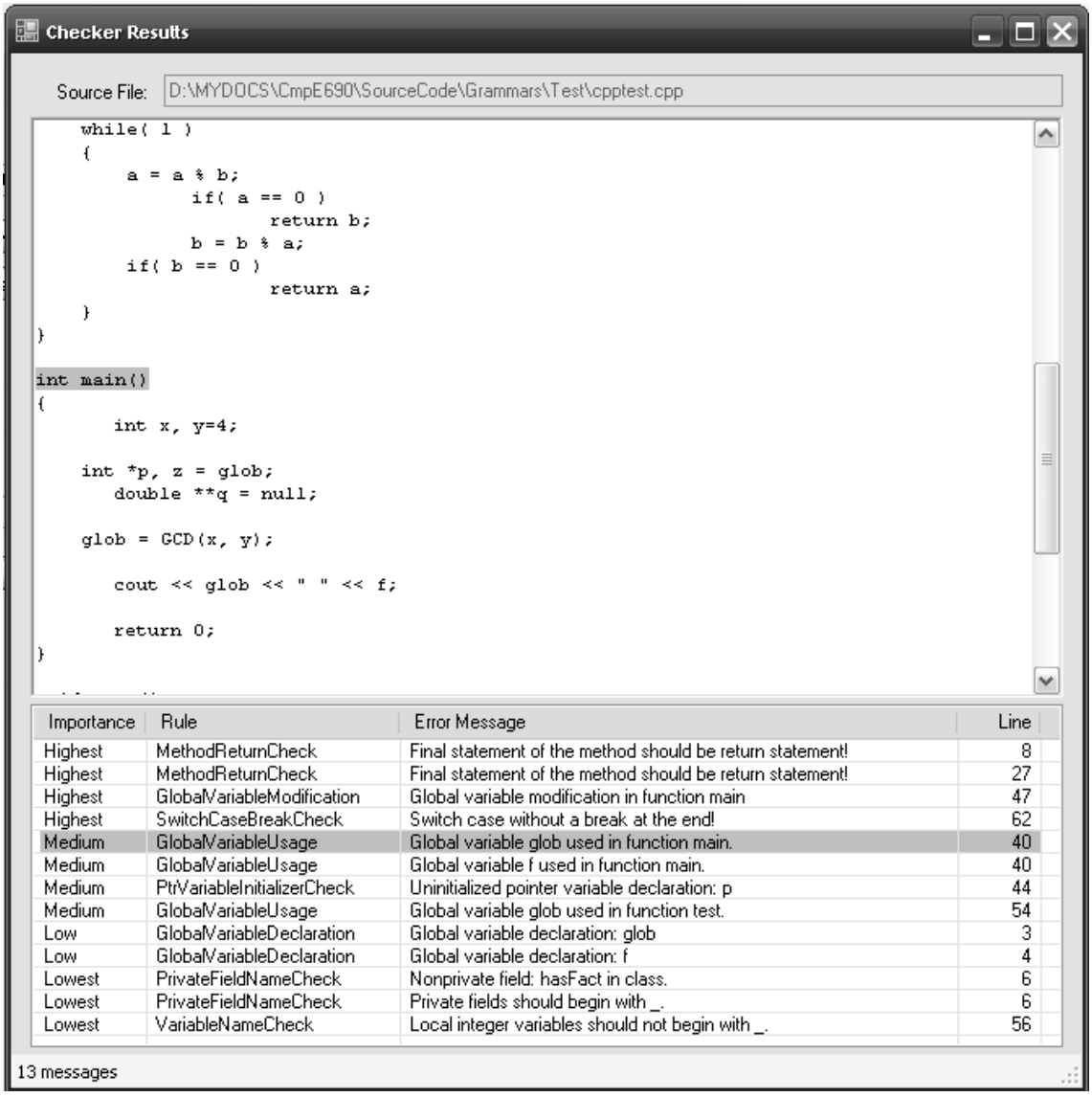

Figure D.3. Sample C++ checker results - III

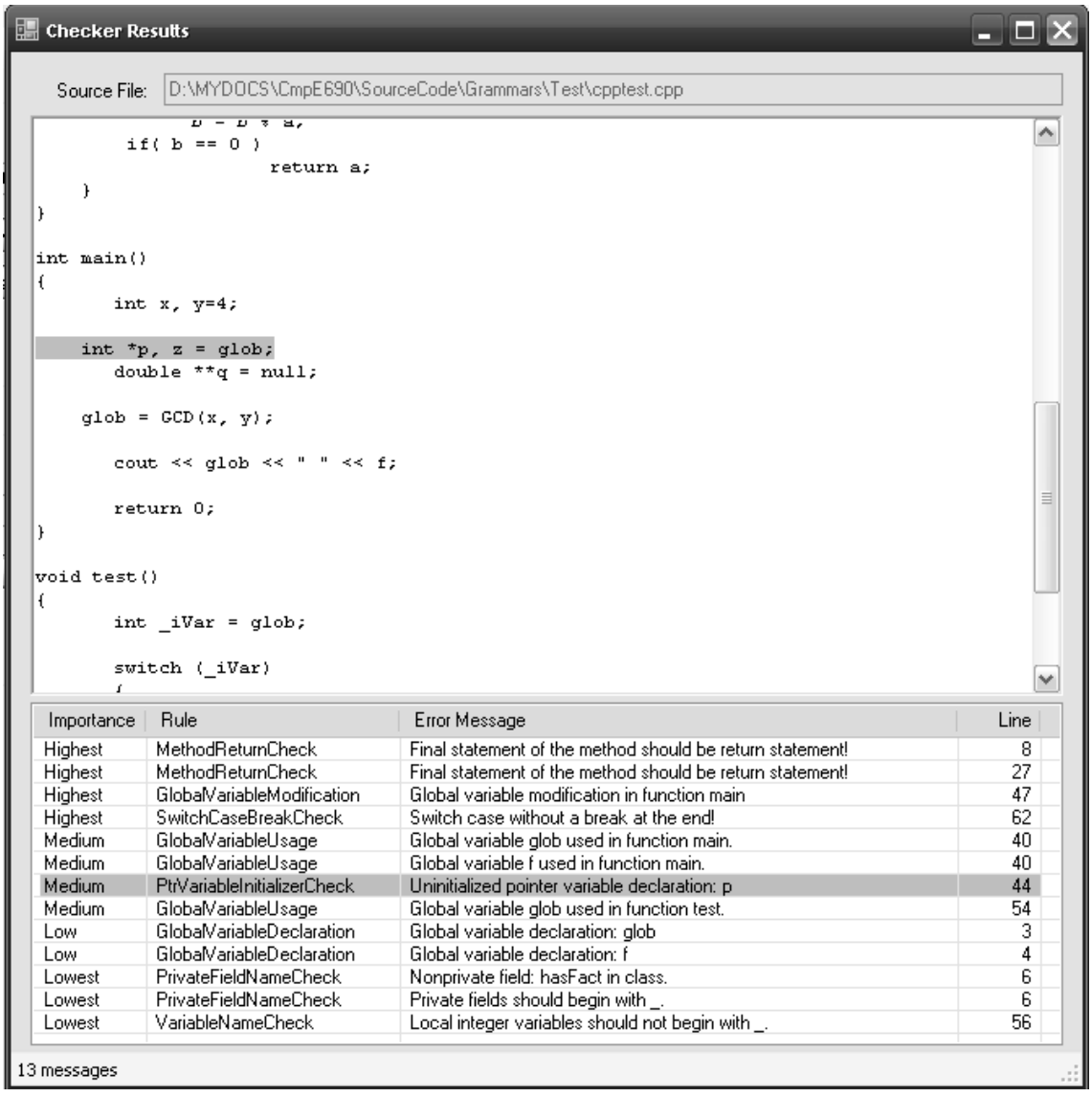

Figure D.4. Sample C++ checker results - IV

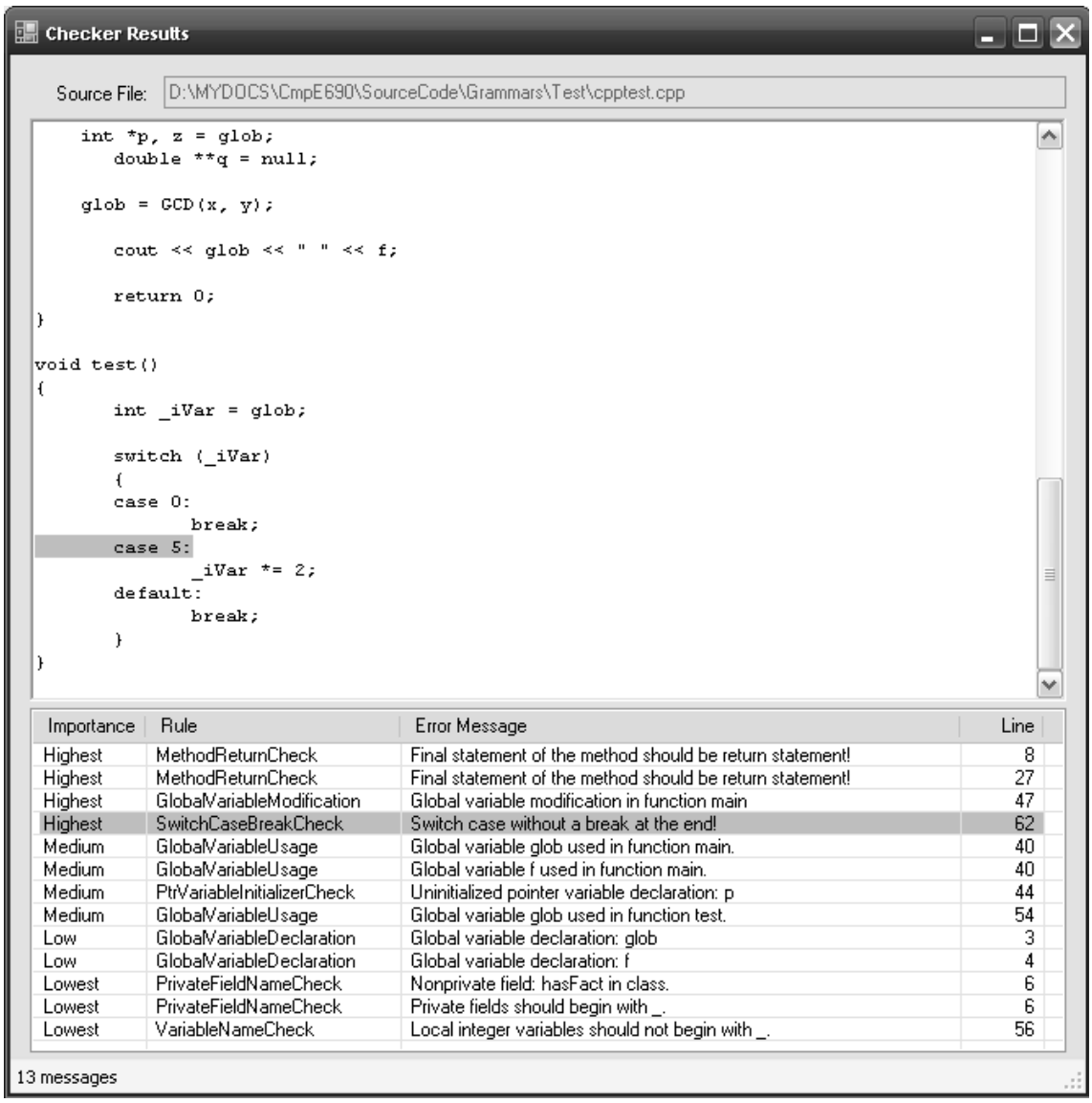

Figure D.5. Sample C++ checker results - V

REFERENCES

1. Louridas, P., "Static Code Analysis", *IEEE Software*, Vol. 23, No. 4, pp. 58–61, August 2006.

2. Ala-Mutka, K., T. Uimonen and H. Järvinen, "Supporting Students in C++ Programming Courses with Automatic Program Style Assessment", *Journal of Information Technology Education*, Vol. 3, pp. 245–262, 2004.

3. Jackson, D. and M. Usher, "Grading student programs using ASSYST", *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, San Jose, California, 27 February-01 March 1997, pp. 335–339, 1997.

4. Holzmann, G. J., "UNO: Static Source Code Checking for User-Defined Properties", *Proc. 6th World Conference on Integrated Design and Process Technology (IDPT2002)*, Pasadena, CA, 26-30 June 2002.

5. Clarke, E., D. Kroening and F. Lerda, "A Tool for Checking ANSI-C Programs", *TACAS 2004 - Tools and Algorithms for Construction and Analysis of Systems*, Barcelona, 29 March-02 April 2004, pp. 168–176, 2004.

6. Potrich, A. and P. Tonell, "C++ Code Analysis: an Open Architecture for the Verification of Coding Rules", *CHEP 2000 International Conference on Computing in High Energy and Nuclear Physics*, Padova, 7-11 February 2000, No. F361, 2000.

7. O'Callahan, R. and D. Jackson, "Lackwit: A Program Understanding Tool Based on Type Inference", *In Proc. 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997, pp. 338–348, 1997.

8. Di Penta, M., M. Neteler, G. Antoniol and E. Merlo, "A language-independent software renovation framework", *The Journal of Systems and Software*, Vol. 77, No. 3, pp. 225–240, September 2005.

9. Reig, F., *Detecting Security Vulnerabilities in C code with Type Checking*, University of California, November 2003.

10. Deeprasertkul, P., P. Bhattarakosol and F. O'Brien, "Automatic detection and correction of programming faults for software applications", *Journal of Systems and Software*, Vol. 78, No. 2, pp. 101–110, November 2005.

11. Depradine, C., "Expert system for extracting syntactic information from Java code", *Expert Systems with Applications*, Vol. 25, No. 1, pp. 187–198, July 2003.

12. Van De Vanter, M. L., "The documentary structure of source code", *Information and Software Technology*, Vol. 44, No. 13, pp. 767–782, October 2002.

13. Jørgensen, M., "Software quality measurement", *Advances in Engineering Software*, Vol. 30, No. 12, pp. 907–912, December 1999.

14. Pollet, I. and B. Le Charlier, "Towards a Complete Static Analyser for Java: an Abstract Interpretation Framework and its Implementation", *Electronic Notes in Theoretical Computer Science*, Vol. 131, pp. 85–98, May 2005.

15. Evans, D., J. Guttag, J. Horning and Y. M. Tan, "LCLint: A Tool for Using Specifications to Check Code", *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, New Orleans, pp. 87–96, December 1994.

16. *Splint (Secure programming lint)*, Inexpensive Program Analysis Group, Department of Computer Science, University of Virginia, http://www.splint.org, 2003.

17. *Microsoft PREfast*, http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx, 2007.

18. *Cxxchecker*, C++ Source-Code Style Check, https://gna.org/projects/cxxchecker, 2005.

19. *System.CodeDom Namespace*, MSDN Library, http://msdn2.microsoft.com/en-

us/library/system.codedom.aspx, 2007.

20. *The C# Language Grammar*, MSDN Library, http://msdn2.microsoft.com/en-us/library/aa664812.aspx, 2007.

21. *Hyperlinked C++ BNF Grammar*, http://www.nongnu.org/hcb/, 2007.

22. *Parasoft C++Test*, http://www.parasoft.com/jsp/products.jsp, 2007.

23. *Programming Research QA C++*, http://www.programmingresearch.com/solutions/qacpp2.htm, 2007.

REFERENCES NOT CITED

1. *MSDN, Microsoft Developer Network Library*, http://msdn2.microsoft.com/en-us/-default.aspx, 2007.

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