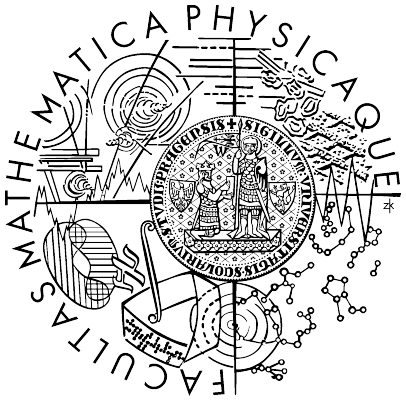
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Libor Olšák

Locating Performance Changes in Source Code

Department of Software Engineering

Supervisor: *RNDr. Tomáš Kalibera, Ph.D.*

Study program: *Informatics, Software Systems*

Computer Science

I would like to thank my advisor, Tomáš Kalibera, for his expert advice he gave me with extraordinary patience and for providing me many helpful consultations.

I hereby declare I wrote the thesis myself, using only the referenced sources.
I give consent with lending the thesis.

Prague, 13 December 2007 Libor Olšák

Table of Contents

Název práce: *Locating Performance Changes in Source Code*
Autor: Libor Olšák
Katedra (ústav): *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Tomáš Kalibera, Ph.D.*
e-mail vedoucího: *kalibera@dsrg.mff.cuni.cz*
Abstrakt:
V regresivním testování výkonnosti je často komplikované najít změnu ve zdrojovém kódu, která způsobila změnu výkonnosti detekovanou dílčím výkonnostním testem. Protože změn je typicky hodně, ruční prohledání všech změn a rozhodnutí, jestli některá konkrétní změna způsobila změnu výkonnosti ve velkých projektech, může být časově náročné.
Práce analyzuje možné metody pro zrychlení tohoto hledání. Zaměřuje se detailněji na dvě z nich. První metoda je mít software, který rozumí zdrojovému kódu a dokáže rozhodnout, které změny mohou potencionálně ovlivnit výkon. Například změny v komentáři výkon ovlivnit nemohou. Druhá metoda je vybrat změny ve zdrojovém kódu, který byl spouštěn během výkonnostního testu. Metody jsou implementovány do Mono Regression Benchmarking Suite jako důkaz funkčnosti návrhu.
V této diplomové práci je rozebráno více možností, jak usnadnit programátorovi hledání a dvě z nich zmíněné výše jsou implementovány.
Klíčová slova: Regresivní Benchmarkování, Výkonnost, Zdrojový kód

Title: *Locating Performance Changes in Source Code*
Author: Libor Olšák
Department: *Department of Software Engineering*
Supervisor: *RNDr. Tomáš Kalibera, Ph.D.*
Supervisor's e-mail address: *kalibera@dsrg.mff.cuni.cz*
Abstract:
In regression benchmarking, it is often complicated to locate the source code modification which caused a performance change detected by a particular benchmark. Manually examining all modifications and resolving if some specific modification has caused the performance change in large projects can be time-consuming.
The thesis analyses possible methods for making that examination faster; focusing in more detail on two of them. The first is to have software that can understand the source language and can resolve which modification can potentially cause performance changes. For example, modifications in comments can not affect performance. The second is selecting source changes that were executed during the test. As a proof of concept, the methods are implemented in Mono Regression Benchmarking Suite.
In this master thesis, more solutions how to increase time efficiency of the search are discussed and the two noticed above are implemented.
Keywords: Regression Benchmarking, Performance, Source code

1. Introduction

Nowadays the complexity of software is rising. Applications have to be developed by large teams to be finished in reasonable time.

Development of some applications is a process, e.g. when one version of the application is released developers are starting to work on a new version by adding or modifying functionality. The time interval between two versions normally is from one month to several years. Developers have much more development versions internally that can be created every day.

In such complex applications it is difficult to maintain functionality and performance during the development process. There are many methods, how to eliminate number of errors that have effect on software functionality. These errors can be detected for example by regression testing [1] or functional testing [2] and they can be located by debugging the system.

It is vital to maintain quality (functionality and performance) of the application version over version. If some code which decreases quality of the application is inserted and this code is not correct, some part of the application can use this code in latest versions. It will be more difficult to correct the wrong code in future because another code may need modification also.

The degradation of performance between two versions can be caused by irrevocable consequence of adding new functionality or oversight of some internal consequences leading to performance degradation.

Some methods like regression benchmarking can detect performance degradation between two software versions, but methods of how to find the cause of the performance degradation are still not very effective.

The topic of this thesis is to develop methods and implement tool that helps a developer to find modification or modifications of the source code that has caused performance degradation.

2. Background

To locate the cause of a performance change it is necessary to detect in what versions the performance change occurs first and this chapter describes methods for the performance change detection.

Regression Testing

The purpose of regression testing is to detect errors in functionality of the application. Regression tests include *unit tests*, which test functionality of small and isolated parts of the application and *system tests*, which test functionality of the whole application.

Regression tests run on synthetic data (often on data - like null and extreme values- , that would more likely lead to application failure or incorrect behaviour). Regression tests are written by authors of the source code or by independent testers. The process of testing is fully automated and developers are notified if some test fails immediately.

The primary purpose of the regression testing is to detect errors in functionality, but it can be used to detect performance changes by measuring response time of the test also.

Benchmarking

Benchmarking is an experimental performance evaluation technique that attempts to measure performance of a computer system under realistic workload. The tested system is exercised by a specialized application called a benchmark, which is designed to simulate a real usage scenario of the system. Values important for the performance - like response time or memory usage - are evaluated statistically because nowadays systems have frequent performance fluctuations.

Regression Benchmarking

Regression benchmarking is a method for locating performance changes between two versions of the software. It is a special application of benchmarking which is tightly integrated with the development process and is fully automated.

Stable set of benchmarks run on daily versions of the tested application regularly. Every version benchmark results are compared with the previous version results. Performance change - improvement or degradation - is detected automatically.

The output of regression benchmarking is normally a list of the performance changes found. All versions with performance changes code modifications are automatically added in this list.

Developer is noticed about degradation of performance as soon as possible and can view all the modifications of the code that were made from the last version with the different performance. Developer can be noticed about improvement of performance also, but improvement of performance is no need to fix.

There are many projects that use automated benchmarking method, e.g. GCC [3], Linux [4], TAO [5], Solaris (operating systems patches) [6]. These projects are focused on tracking performance of the particular projects. But these projects are lack of changes automatic detection as well as finding performance alternation causes.

Mono Regression Benchmarking Project

Mono Regression Benchmarking Project [7] is an implementation of the regression benchmarking in Mono [8] project.

Mono is an open source implementation of Common Language Infrastructure specification [6], known as the .Net platform. The Mono implementation of CLI comprises a C# compiler, a virtual machine interpreting Common Intermediate Language instructions, and an implementation of the runtime classes.

Mono Regression Benchmarking Project is automated, from downloading daily Mono versions, through benchmarking and analysis, to visualisation of the results. The results are available on the web [7] covering daily Mono versions since August 2004.

The project uses several benchmarks for different Mono parts. To increase coverage of the Mono project by the benchmarks, Mono Regression Benchmarking Project also runs Mono regression tests as benchmarks (currently about 100 tests).

Performance changes are presented in two ways: The first is *Changes Summary Chart* (Figure 2.1) that is a table with the performance changes summary of all supported benchmarks on all platforms. This chart contains seven latest Mono versions only.

	2007-08-21		2007-08-20		2007-08-19		2007-08-18		2007-08-17		2007-08-16		2007-08-15	
	P4	IA64	P4	IA64	P4	IA64	P4	IA64	P4	IA64	P4	IA64	P4	IA64
HTTP Ping	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A
HTTP Ping (all JIT optimizations)	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A
ICP Ping	=	N/A	=	N/A	=	N/A	=	N/A	-19.33%	N/A	22.59%	N/A	=	N/A
ICP Ping (all JIT optimizations)	=	N/A	=	N/A	=	N/A	=	N/A	-19.91%	N/A	23.59%	N/A	=	N/A
FFT SciMark	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A
FFT SciMark (all JIT optimizations)	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	3.02%	N/A
Equake	=	N/A	=	N/A	=	N/A	-3.01%	N/A	=	N/A	=	N/A	=	N/A
Equake (all JIT optimizations)	=	N/A	=	N/A	=	N/A	2.33%	N/A	=	N/A	=	N/A	=	N/A
FFT SciMark (no allocation)	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A
FFT SciMark (no allocation, all JIT optimizations)	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A	=	N/A

Figure 2.1: Summary chart

The purpose of this chart is to allow developers to check if the performance

changes were detected from the latest versions, how significant was the change and which platforms and benchmarks were affected.

The second is a *Detailed Changes Chart*. It contains a table and a graph of changes detected by each of the benchmarks. This chart includes the full history of Mono versions. It allows developers to observe long-term performance trends.

Part of the project is source code diff, which shows changes between two versions, where performance change was detected. The changes can be reduced to class libraries sources that are potentially used by the concrete benchmark.

3. Goal of the Thesis: Locating Causes of Performance Changes

Regression benchmarking is a method that gives to developers a software version where performance has been changed. The problem is that too many changes can be made in source code of this version and it may be difficult to locate changes that affected performance.

It is hardly possible to find the cause of performance degradation in the source code automatically. Many consequences can be in source code. These consequences are problem for automatic detection of the cause.

For example: Execution of some method can change some internal value of the object that affects execution of another method.

The primary goal of the thesis is to develop general methods that are platform and programming language independent and that would help developers to locate the performance degradation cause in a reasonable time. These methods operate with a source code of two versions of one application as input: the versions where performance change was detected. As output these methods produce the reduced list of changes between these two versions. This list contains changes that could affect application performance and allows the developer faster orientation in these changes.

The secondary importance goal is to implement these methods as a .Net console application. The tool will recognize syntax of source code written in C# language [9] in version 2.0.

Output should be generated HTML pages with detected modifications or the output will be written to console. The console output should be possible to be used as input in another application.

The tool will be tested on some existing system to evaluate how effective it is.

Because of several reasons the Mono became an application of choice for the testing. The first reason is Mono Regression Benchmarking Project which is implementation of regression benchmarking. This implementation allows application testing by using benchmarking results and finding performance degradation cause. The second one is that the described in thesis tool can be added as part of Mono Regression Benchmarking Project easily. The last but not the least are Mono complexity and size (source code of Mono contains over 2 million lines), active project development and open-source licence. Results of the testing will be described in the Chapter 8.

4. Structure of the Thesis

The thesis text is organized as follows:

Chapter 5 describes existing methods of finding the cause of performance degradation and proposes new, more effective methods that solve that problem.

In Chapter 6 more details language sensitive diff with traced analysis is introduced.

Chapter 7 describes implementation of the ContextDiff. The chapter contains installation and user guide as well as architecture of the ContextDiff.

In Chapter 8 ContextDiff is tested on different version of Mono. Generated diff is compared with the existing diff and then their efficiency is compared. The other part of the chapter gives the reason of the performance change.

Chapter 9 contains conclusion with the description of possible future work.

5. Analysis: Semi-automating Location of Causes of Performance Changes

When application version-to-version performance change is detected, developers usually compare the source code of these versions.

For this purpose all the versions have to be stored somewhere. This need is met by using special *version control* applications, e.g. Subversion [10] or Concurrent Versions System [11]. Version control applications store all the versions of the developed application on server. They allow assigning comments to each version and they logs dates of each creation of the version and author of the version.

Version control applications have some basic functions that can be used to locate the cause of the performance changes, but using them for this purpose is not very effective. They allow developer to compare source code files by *classic text diff* or browse comments of the changes.

5.1. Classic Text Diff

Classic text diff compares the same source code file from two versions of the same application but it cannot recognize syntax of the programming language. Diff works with pure text without additional information - like type of text - only. All the character types (e.g. alphanumerical characters, spaces, tabulators, comas ...) except the *newline* character have the same importance.

Text diff compares lines of the source code files. If any character (including spaces and tabulators) of the line does not match character on the same position in the second line, both lines are considered as not equal. The text diff marks lines, about which it decided, they are deleted, inserted or modified. But need to mention that there is no absolute criterion how to recognize, which lines are modified.

Example: Some method in older version contains only one row “Execute(X);”. The same method in newer version contains two rows “Execute(Z);” and “Execute(W);”. Recognize which one is the original row and which row was added is in this case not possible.

Some text diffs do mark no line as modified ever. Each line that is modified is in source code of older version marked as deleted and in the same line in source code of newer version marked as added.

When more lines are changed consecutively, some lines can be inserted or deleted to/from this changed block. Some diffs could mark whole shorter block (changed block in version where is shorter) of changes and first N lines (where N is length of the shorter block) of the longer block as changed. Remaining lines from longer block are marked as inserted (if they are located in newer version of

the source code) or deleted (if are located in older version of the source code).

Text diff can also have user-friendly (e.g. graphic) output to offer the developer better orientation in changes. The output usually contains two columns: the left one with the former version code and the right one with the newer version code. Lines of source code that correspond between versions are on the same row on the output. Deleted, inserted or modified lines can be recognized easily as marked by different colours.

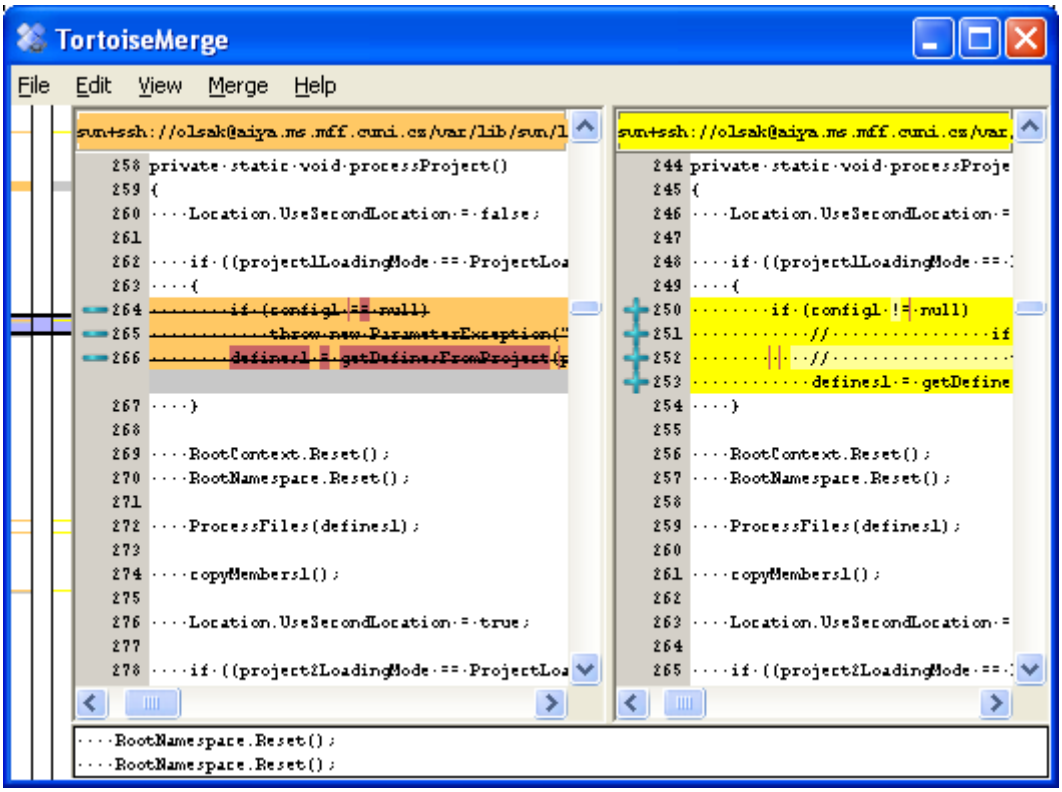


Figure 5.1: text diff integrated in Subversion.

Figure 5.1 Shows a TortoiseSVN integrated diff as an example. This diff do not recognize modified lines: all the modified lines are marked as deleted in the former version or as inserted in newer version of the file.

Text diff can be used for many purposes like code insertion verification, bugs finding and new code inquire.

Developers use also text diff on the basis of experiences for locating source of detected performance change. Results from more benchmarks on the same configuration and benchmarks on different configurations and platforms can be used to help them to find the cause of the performance change.

5.1.1. Problems

Text diff has some restrictions that can affect its usage in order to detect

performance changes.

The first problem concerns the source. The code can be very imposing and many developers can work on the application at the same time. Sometimes it can be difficult to understand the source and these complications can also slow the work. The developer has not to focus on changes only but to read and understand code around changes also. It means that the person who actually works on the code needs to investigate methods that are changed and methods that call changed methods properly or are called from changed methods.

In order to simplify developing of such a complex application it's strongly recommended to add comments to the source.

The second problem is large number of changes that occurred from one version to another because verifying big number of changes could take a lot of time. A good example is a Mono project diff sizes Figure 5.2.

Date from	Date to	Changed lines	Added lines	Deleted lines	Performance change
2006-11-02	2006-11-13	5100	16759	1846	-7.89%
2006-10-04	2006-10-05	983	3776	1911	+2.05%
2006-04-19	2006-04-20	311	935	536	-2.37%
2006-02-27	2006-03-17	4426	53997	3293	-9.02
2006-02-08	2006-02-12	2278	5220	2214	+3.82
2006-02-03	2006-02-04	179	2253	431	-1.63

Figure 5.2: Diff size of Mono project.

Source: http://dsrg.mff.cuni.cz/projects/mono

The next problem is the source code formatting. It is possible to set up style of formatting in many source code editors (for example in Microsoft Visual Studio 2005 [12]) but problem can occur when not all the developers working on the same project and have different formatting habits. That's why a lot of unwanted formatting changes can be added while editing source by another developer. Diff doesn't consider text with a different formatting (that is irrelevant to the functionality) as the same and displays these changes like any other significant change. That's why it makes more difficult to find parts of the code that can really change functionality.

The fourth problem is lines that are inserted right after or before the line modified: text diff may not recognize them. The reason is in a diff internal logic – diff have only two modes for lines, equal or not equal. So if there is only one changed character in line the line has the same flag (not equal) as the line that is completely different. This point complicates orientation in changes also.

Example (Figure 5.3): Green colour represents the line that was modified and blue colour represents added line. Line 104 in older version was expanded by tabulator character and before this line new line was added. Added line is marked as modified and modified line is marked as added.

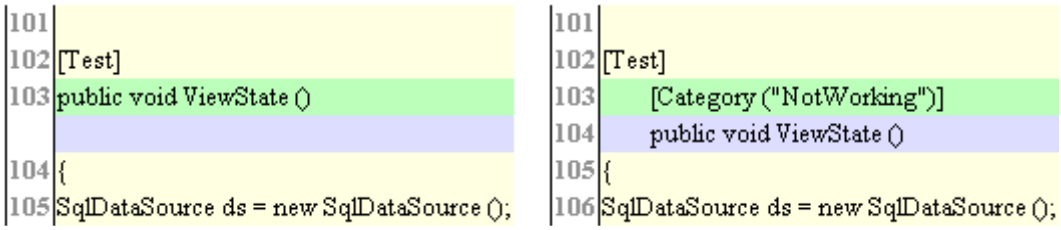


Figure 5.3: Wrong marked modified and added line.

5.2. Diff Visualization Improving

Visualization of the diff results is very important. Existing text diff can render compared source code files to two columns, where corresponding lines are rendered on the same row (it was described in Chapter 5.1).

Presentation of diff results can be improved by using programming language syntax knowledge.

5.2.1. Syntax Highlighting

Syntax highlighting is a feature that displays text in different colours and fonts depending on the category of the displayed term. Each term from one category is always rendered with the same colour. The number of categories depends on programming language used.

Syntax highlighting is offered by source code editors like Microsoft Visual Studio [12], Eclipse [13], Net Beans [14] and many others. The feature eases writing in a structured programming language. Syntax errors are visually distinct.

Diff tools work with the text files only and do not have information what type of the text files they are – source code files, configuration files written in XML format, documentation file or some other file.

Necessary to mention that it's important to know what programming language is used for the code writing in order to highlight the source. Highlighting is important on keywords, comments, strings and conditional compilation.

Keywords

Keyword is a word that has particular meaning for the programming language and this meaning of the keyword can differ from language to language. In languages, such as C, C# or Java, keywords are reserved words and cannot be used as names of variables, functions or types. Some languages, such as PostScript, allow keywords to be redefined for specific purposes.

Keywords are important for functionality and performance of software. Highlighting of keywords is vital for better orientation in a source code.

Comments

Comments are a part of the source code that has no effect on functionality. Comments have a wide range of use: they can be used for code description, for writing notes and for generating external documentation. Sometimes comments can be used for some code parts disabling instead of deleting. In case of temporary disabling of the code it's necessary to have commented code highlighted. Otherwise comment can be overlooked easily.

String Constants

String literals represent string values in a source code. Strings can contain any sequence of characters and are bounded in source code with special characters, typically by apostrophe or double apostrophe depending on programming language. They can also contain keywords but the meaning of that word is not taken into consideration.

Strings should be highlighted also because they can contain sequences of characters that look like a source code but are not a source code.

Preprocessor Directives

Preprocessor directives are parts of source code used by Preprocessor. Preprocessor is executed before the actual compilation of the code begins; therefore preprocessor digests all preprocessor directives before any code is generated. It can be used for example to define macros, issue errors and warnings or to mark blocks of source code for *conditional compilation*.

Conditional Compilation

Conditional compilation is a feature implemented in many programming languages. It allows the developer to define during the compilation what parts of the source will be compiled and what will not. It can be useful for example for multi-platform compilation.

Parts of the source code that are not compiled are usually highlighted with some featureless colour, normally grey.

To recognize, which part of the source code is compiled and which part of the source code is not, settings of the compiler are needed to know.

```
#if NET_2_0 || BOOTSTRAP_NET_2_0
    // Versions of .NET Framework 2.0 RTM
    public const string FxVersion = "2.0.0.0";
    public const string VsVersion = "8.0.0.0";
    public const string FxFileVersion = "2.0.50727.42";
    public const string VsFileVersion = "8.0.50727.42";
#elif NET_1_1
    // Versions of .NET Framework 1.1 SP1
    public const string FxVersion = "1.0.5000.0";
    public const string VsVersion = "7.0.5000.0";
    public const string FxFileVersion = "1.1.4322.2032";
    public const string VsFileVersion = "7.10.6001.4";
```

Figure 5.4: Syntax Highlighting of Source Code

Example of syntax highlighting is shown on Figure 5.4. Keywords and preprocessor directives are rendered with blue colour. Comments are green and string constants are red. Part of the code that is not compiled due to conditional compilation is grey.

There are many groups that can be highlighted but just some of them are used in order to keep the code readable.

Example: Visual Studio 2005, 112 groups…

5.2.2. Hyperlinks

A hyperlink is the next way how to improve readability of the diff results.

Hyperlink is a document navigation element that points at another part of the same document or to another document. User can simply click on the hyperlink and browser will automatically open linked document or scroll to the target part of current document.

In diff hyperlinks can be used to navigate developer to modified methods or classes. All modified methods and classes can have hyperlinks in displayed source code instead of static text. These hyperlinks lead to the bodies of the linked methods.

This method has a disadvantage: It is not always single valued, which method is called from some concrete position in source code, from static analysis. More methods that can be from specific location called in one execution of the application can exist.

Example: Consider some interface with headers definition and its method. Every class that support the interface has to implement all the methods from this interface. One interface can be supported by several classes as well as one class can support several interfaces. Classes can work with interfaces without knowing class that implements the called interface.

5.3. Code Profiling

The next method of how to locate the cause of the performance change is so called profiling.

During the application execution not all of the parts of the program have the same processor time .There are several method types that can help to solve that problem – "short methods" (with a short and rare execution) and "long methods" (methods that are being executed many times or contain some time consuming algorithm). Optimizing "short methods" bring only small performance increase.

Profiling is the method that finds "long methods" during the tested application execution.

Profiling is similar to benchmarking, but profiling collects more performance data about running application and is used on real application and not on some model.

Profiling can extract every method execution time, memory usage statistics, code coverage data, trace calling and thread information of each executed method. Because collecting big amounts of data is a time consuming process, profiler can (but not always do so) affects the application execution time by slowing down the system. Performance results can be incorrect in absolute values but not necessarily in relative values.

There are many profiling tools that were implemented in different languages, e.g. IBM Rational Application Developer [15]. *Execution Statistics* (Figure 5.5) and *Performance Call Graph* (Figure 5.6) features of the tool are important for improving performance.

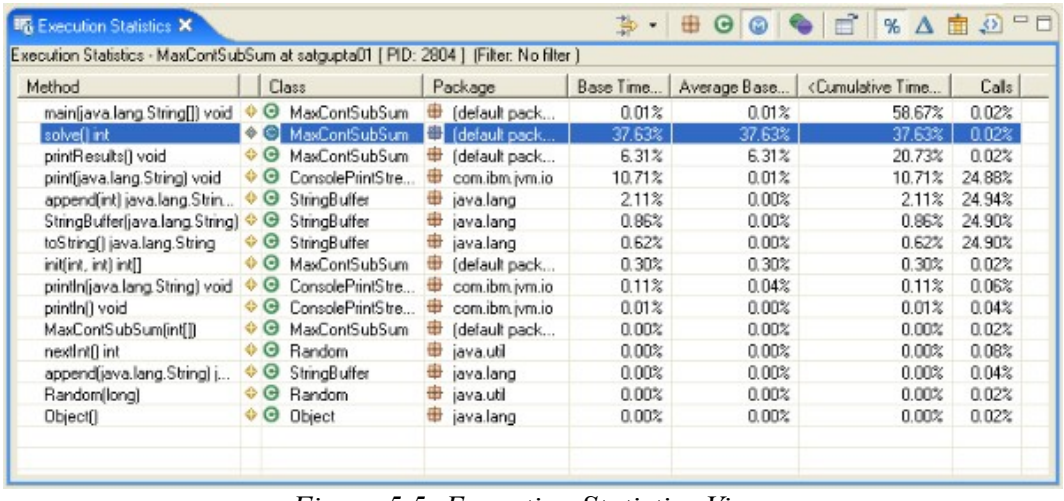


Figure 5.5: Execution Statistics View.

Execution Statistics view shows information about running time of each method.

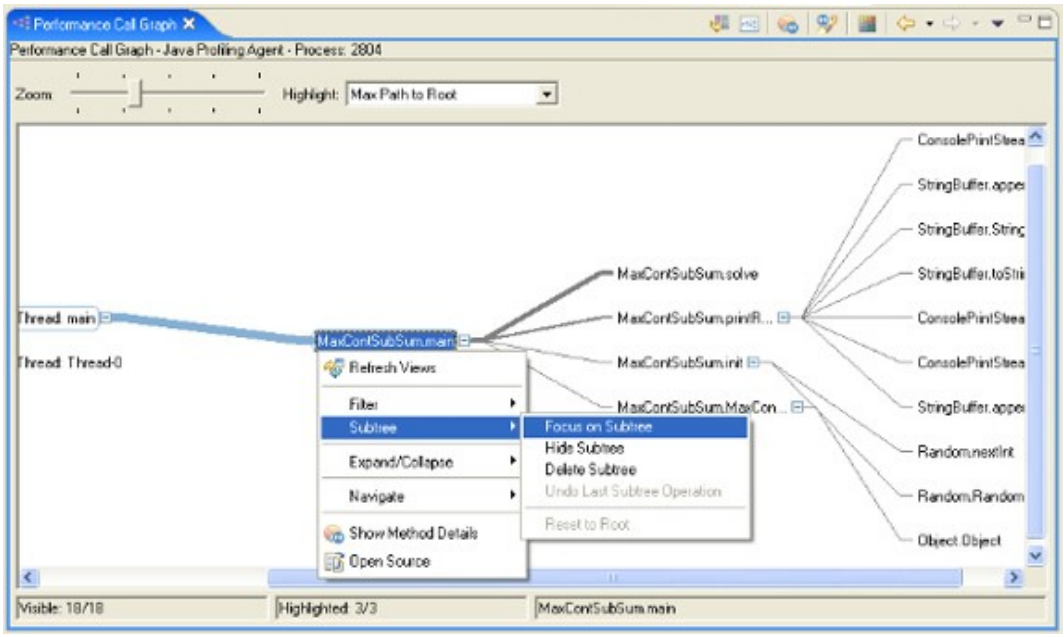


Figure 5.6: Performance Call Graph View.

Performance Call Graph shows call tree which contains the most time-consuming nodes (node represents a method, a process or a thread). Thicknesses of line denote how time-consuming node is.

Profiling is typically used to improve performance of the application final version but can be also used as one of the methods of the performance change cause location. The cause can be indicated by methods of the two different version running time comparison. If the one founds that some method is “slower” than the one in the previous version, he will try to increase the method performance; if the one founds that some method is executed more often than in previous version, he will try to decrease number of the method executions.

But application performance could be changed by another parts of application modification. That “cause” modification is difficult to find using profiling only and other methods should be applied also.

When performance of some method is decreased, all methods that execute mentioned method could be affected. Therefore it could be useful to compare call trees of both versions and find decrease of performance on the lowest level of the both trees.

The problem is the call trees may not have the same structure between two calls of the same application because other methods are can be called during execution of the application. It can be affected for example by state of the cache. Distinction between the call trees can be higher between executions of two versions of the application and it can slow down or disable locating cause of performance change with use of profiling.

5.4. Reducing Changes to Review: Code Analysis

There are several groups of changes that cannot have effect on performance and that are possible to recognize automatically. Speed of the performance change cause locating can be also improved by removing such changes from the displayed list.

To recognize these changes it's necessary to have some knowledge of the programming language syntax and the compilation process. Therefore text diff cannot reduce the displayed list of changes.

There are two basic groups of changes that do not necessary have effect on performance. The first one contains changes that don't affect the performance generally. The second one contains changes that don't affect concrete benchmark performance.

5.4.1. Static Analysis of Source Code

Source code static analysis is the analysis that is performed without executing program built from this source code [16].

This method is often used in areas such as software validation, software re-engineering or validation of software security [17], but the method can be also used to locate changes that do not necessary have effect on performance.

Typical changes that don't affect performance are changes in comments, white-spaces (but not in string constants), position of methods or classes in file or in more files, objects accessibility changing and renaming of methods and classes.

Some of these changes can affect the performance of the application in the concrete programming language. For example position of executed methods in memory can be affected by positions of the methods in a source code files or by the names of these methods. Methods positioning in memory can affect CPU cache and can have effect on performance.

Many programming languages allow the developer to use conditional compilation when he or she can define by using *preprocessor directives* which parts of the source code will be compiled and which parts of the source code will be not compiled. Parts which are not compiled cannot have effect on performance.

5.4.2. Analysis of Traces

Changes in methods that were not executed during concrete benchmark cannot affect performance of this benchmark. These changes can be removed from displayed diff.

Figure 5.7 displays the amount of changes of the same versions of Mono project as in Figure 5.2, but changes in Figure 5.7 are reduced to the class libraries sources that are potentially used by HTTP Ping benchmark [7]. The amount of changes decreased rapidly.

Date from	Date to	Changed lines	Added lines	Deleted lines	Performance change
2006-11-02	2006-11-13	35	79	9	-7.89%
2006-10-04	2006-10-05	1	0	0	+2.05%
2006-04-19	2006-04-20	1	0	0	-2.37%
2006-02-27	2006-03-17	230	274	17	-9.02
2006-02-08	2006-02-12	30	9	8	+3.82
2006-02-03	2006-02-04	96	152	126	-1.63

Figure 5.7: Reduced diff size of Mono Project.

Source: http://dsrg.mff.cuni.cz/projects/mono

It is difficult to find methods which are executed during the benchmark without execution of the application. Therefore these methods have to be found during execution of the application. Method that logs information about the methods execution is called tracing and its output is called trace.

There are several ways, how to get the trace. Some platforms like Microsoft .Net [18] allow retrieving trace information without any application source code change and without recompiling the application. In some system it is necessary to recompile application with trace parameter or modify the source code.

Combination of Benchmarks

Combination of more different benchmarks created from the same application can reduce number of displayed changes. If some benchmarks detected decrease of performance and if other benchmarks detected no performance change then it means that the cause is in the code that was executed in all benchmarks that detected decrease of performance.

In many cases, the change in the source code that caused performance degradation affected execution of the method, where the change is located. In these cases, if some particular benchmark detected no performance degradation, modification of the source code executed by this benchmark can not affect performance. These modifications can be removed from the displayed list of changes.

But in some cases the change of source code could affect application performance on another part of the program. In these case changes of the code that was executed by benchmarks that detected no performance degradation cannot be removed from the displayed list of changes. Example of this case - change in the method that affects inner state of some object and can affect performance of another method.

It is not predictable which case of performance change it was. Therefore changes in the code that was executed during benchmarks that detected no performance change cannot be removed from the list of displayed changes, but can be marked as less probable cause of a performance change or can be offered

to review to developer in the second step, if in the first step was the cause not located.

Combination of benchmarks described above works if there is only one cause of performance change in the source code. If benchmarks are executed frequently (for example on daily basis) it would be probable that if performance change is detected the cause would be only in one modification of the source code. Therefore it is useful to highlight changes detected by a combination of benchmarks or display them in the first step of locating the cause and in the second step count with more causes of performance change.

5.5. Quantifying Potential Performance Implications of Code Changes

Some changes in the source code have lower probability of application performance affecting. For example, string of constants is often used to show user of the application some message and the text of this message can not affect performance.

But the string of constants can be also used to control the application execution. In this case changing such a constant can affect functionality and performance of the application.

Potential performance implication of the source code changes is dependent on programming language but also on the concrete application.

If developers knew that string constants were used in the application for displaying message to user and logging only, then changes in string constants could be considered as improbable cause of the performance change. This change could be less highlighted in diff or could be shown in some later step of finding the cause of the performance change.

If each application step is driven by string constants and application has no user output and logging, changes in string constants are almost always critical to functionality of performance. These changes should be highlighted in diff or should be shown in the first step of finding the cause of the performance change.

This problem can be resolved by adding some mark to these constants (if constants were critical to performance or functionality) or by adding some marks to the parts of the source code, where all constants are or are not critical to the performance or functionality. It is important for the developer to pay attention to these marks and control it.

It is possible to find out which kind of string is used by using static analysis. The string can be used as a function parameter and show some message or be used in some condition. The same problem as described in Chapter 5.2.2 can occur. The constant can be used as a parameter in some function and it can be impossible to find out what function it is using static analysis

6. Solution: Language Sensitive Diff with Run-time Trace Analysis

The secondary goal of the master thesis is to implement *Language Sensitive Diff*. Language Sensitive Diff is a tool that helps developers to locate the cause of performance change by combination of language sensitive diff with reduction to executed code and better presentation of found modifications.

The tool is diff that works with abstractions on programming language level. It allows the diff to work with the whole project and not only with single files and compare objects of the programming language.

6.1. Comparing two Versions of the Source Code

It is necessary to compare both versions of the source code with each other to create the diff. Sophisticated algorithm is required to compare two versions on the abstraction of the programming language level.

The worst case of comparison methods is to compare each method of the older version with each method of the newer version. It is time consuming. Comparing two methods have complexity MxN. When comparing two equal methods, comparison can be done in linear time (described later). When comparing method from the older version with the same method from the newer version of the source code, the most frequent case is that the method is unchanged (comparing two equal methods). The frequency is rising with frequency of comparison. In daily versions are developers able to modify only small part of the source code. It is effective to reduce comparing of the method bodies only to the same methods between the versions.

To do this, it is required to compare source code on all levels of the programming language abstraction. In step one, all corresponding pairs of classes and structures from both versions of the source code are found. In step two, all corresponding pairs of the inner items of the classes (like methods, enumerators and nested classes) are found in corresponding pairs of the classes. In step there, all corresponding method bodies are compared.

Of course, some exceptions in this rule exist.

Not all objects of the programming language have to be inside of any class or structure (like enumerators). The step one is skipped for these objects and they are compared as if they were in one global class for each version.

Classes can contain nested classes. These classes are compared recursively. All nested classes in corresponding pair of classes are compared like in step one.

6.1.1. Comparing Classes

In the first step is needed to find, which classes (and structures) from the first version of the source code correspond with which classes of the second version of the source code.

The first, full names of the classes are compared. If the names of the pair are equal, classes are declared as corresponding.

In the next step, renamed classes are found. All objects inside the remaining classes between both versions are compared (only heads (not bodies) are compared for methods and properties). The *similarities* of the classes are evaluated. The similarity is defined as rate of equal items inside the pair of classes to total amount of items in the class (amount of items is taken from the smaller of the pair of the classes). The classes are declared as corresponding, if the similarity is higher than the pre-set constant. If some conflict occurs (one class from one version can be similar to more classes from the second version of the source code), more similar pairs are chosen.

6.1.2. Comparing Class Items

In the next step, items inside the corresponding pairs of classes are compared.

The names of the items are compared for all items of the same type (methods, constants...). Next comparison depends on the type of the item and on implementation for concrete programming language (each programming language can have specific structures)

For example: For methods, parameters and return value are compared. For constants, value and type are compared.

Renamed methods (in this paragraph; constructors, destructors and properties belong to methods too) or methods with changed parameters are found in the next step. *Similarities* are evaluated for all pairs of methods between both versions of the source code. Similarity for methods is defined as the rate of the length of the longest common subsequence (LCS – see Comparing Method Bodies 6.1.3) to the length of the shorter method of the pair. Pair of the method is declared as corresponding, if the similarity of the methods was higher than a pre-set constant. If the conflict occurs, the pair with higher similarity is chosen.

6.1.3. Comparing Method Bodies

Method (and also constructor, destructor and property) bodies are compared in last step.

Method bodies are compared on the *statement* level. Statements are atomic parts of the method bodies. In the C# source code, statements are separated by semicolon character except conditional and loops.

The first have to be found the unchanged statements and then the modified

statements in the statements, which was changed. The remaining statements are new inserted or deleted (it depends on version, where they are).

While finding corresponding classes or methods between both versions of the source code, position of the classes or methods in files is not important. While finding corresponding statements in corresponding methods, position in the file become important. Changing order of statements generally has effect on functionality and performance of the application.

Longest Common Subsequence

To find corresponding statements, it is necessary to find the *longest common subsequence* (LCS) [19] in ordered lists of statements.

Subsequence of some sequence is the list of objects chosen from the sequence with the same relative position as in the original sequence.

Common subsequence is the subsequence which is a subsequence of all sequences (in this case two) in a set of sequences. Longest common subsequence is the longest of these common subsequences.

Two sequences can have more than one longest common subsequence. For example sequences “abcd” and “acbd” have longest common subsequences “abd” and “acd”. Only one of these longest common subsequences is used to maintain lucidity of displayed modifications (no alternatives are shown).

Finding the longest common subsequence is NP-hard in general case of an arbitrary number of sequences. When finding the longest common subsequence in two sequences (it is the case of comparing two methods), the problem can be resolved in complexity MxN (multiplication of lengths of the sequences).

Finding the Longest Common Subsequence

LCS can be found by using dynamic programming. To do this, it is necessary to apply recursive solution. The solution can be found from the following observations:

Common subsequence (not only the longest) can be represented as a way of writing two strings so the certain letters line up. Example for two string “Regression testing” and “Performance” is shown in Figure 6.1.

Figure 6.1: Lined up letters.

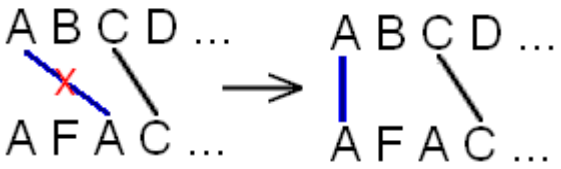
If lines are drown between letters in the first string to the corresponding letters in the second string, no two lines can cross (the top and bottom endpoints

occur in the same order that is the order of the letters in the subsequence). Conversely any set of lines drawn like this (without crossings) represents a subsequence of the two strings.

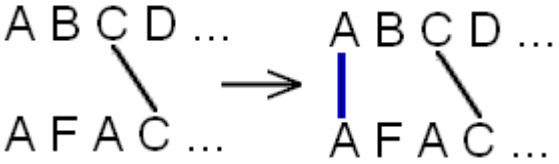
If two strings start with the same letter following possibilities can occur:

1. The letters are connected with line. The character is part of the subsequence.

2. One of these two letters is connected to the letter that is not the first letter of the second string. Then this connection can be broken and the first letters of both strings can be connected. No other line will be crossed, because the letters are the first in the strings. The subsequence will be the same (created from another instances of letters but with the same value).



3. None of these two letters is connected to any letter of the second string. Then longer subsequence will be created by connecting these two letters.



4. The last possibility that the both letters are connected to any (not the first) letter of the second string cannot occur, because the lines in this possibility cross each other.

In this case (two strings start with the same character), the first letter can be chosen as the first character of the longest common subsequence.

If two strings do not starts with the same letter, than it is not possible that both letters are parts of common subsequence, because if they are connected to any letter (not the first because they differs) of the second string, the lines will cross each other. The longest common subsequence is in this case equal to longer of following two sequences. The first is longest common subsequence of the first string unchanged and the second string with removed the first character. The second sequence is the longest common subsequence of the first string with removed the first character and the second string unchanged.

These observations can be summarized to this recursive rule:

Given the sequences $x_{1..m}$ *and* $y_{1..n}$

$$LCS(x_{1..m}, y_{1..n}) = \begin{cases} \emptyset & \text{if } m=0 \vee n=0 \\ x_1 + LCS(x_{2..m}, y_{2..n}) & \text{if } x_1 = y_1 \\ max(LCS(x_{1..m}, y_{2..n}), LCS(x_{2..m}, y_{1..n})) & \text{otherwise} \end{cases}$$

For equal sequences is $x_1=y_1$ in each recursion (except the last recursion, when both sequences are empty). Finding the LCS for equal sequences can be done in a linear time.

The result of each particular call of LCS recursion can be stored in the table. In the worst case, the MxN table is filled (complexity MxN).

Modifications

Longest common subsequence finds sequence of statements that is contained in both compared method bodies. Remaining statements are split into statements that are modified, statements that were inserted in newer version and statements that were removed from older version. It is necessary to find modified statements only. The remaining statements are inserted, if they were in the newer version, or deleted, if they were in the older version.

When removing longest common subsequence from compared method bodies, several substrings* of statements remain in both method bodies. Modifications are found only between *corresponding substrings*.

Substrings between two versions of the method are corresponding, if the statements right before the substrings were corresponding or if no statements were

*Substring of some sequence (or string) is the consecutive part of the sequence. Substring is always subsequence but subsequence has not to be substring. For example: Let "abacadaeafag" be a sequence, then sequence "cadaea" is substring of the original sequence, but sequence "cdefa" is only subsequence and is not substring of the original sequence.

before both substrings (both substrings are in the beginning of the methods).

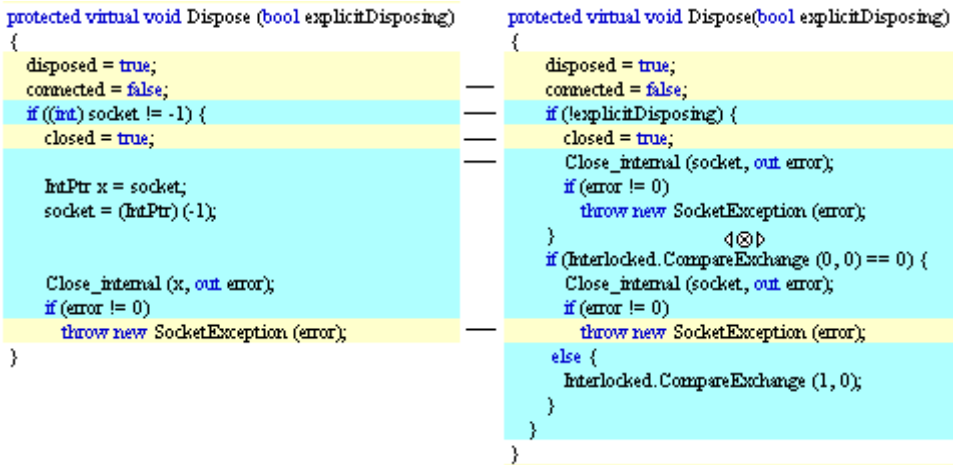


Figure 6.2: Longest common subsequence and corresponding substrings.

On Figure 6.2, two versions of one method are displayed. Longest common subsequence is marked with yellow colour and remaining substrings are marked with blue. The first two pairs of the substrings correspond because statements right before them corresponds. The last substring in the right column has no corresponding substring in left column, because no substring is below statement that is corresponding statement two statement right before the substring.

The same LCS algorithm can be used again to find modified statements, but the rule is slightly modified.

Given the sequences $x_{1..m}$ *and* $y_{1..n}$

$$LCS(x_{1..m}, y_{1..n}) = \begin{cases} \emptyset & \text{if } m=0 \vee n=0 \\ x_1 + LCS(x_{2..m}, y_{2..n}) & \text{if } x_1 \simeq y_1 \\ max(LCS(x_{1..m}, y_{2..n}), LCS(x_{2..m}, y_{1..n})) & \text{otherwise} \end{cases}$$

The second condition is modified from $x_1 = y_1$ (are equal) to $x_1 \simeq y_1$ (are similar). The statements are similar, if statements are of the same type (condition, assign, for loop, while loop...) and some of their parts (depends on the type of statement) are equal.

It is not possible, to detect verily if the statement was modified or if the statement was deleted and new statement was inserted; or if more modified and inserted statements were in the same substring, which are modified and which are inserted. The detection can only guess by the similarity of the statements.

6.2. Reducing the Diff Size

The tool reduces the size of the diff as described in chapter 5.4 using static

analysis of the source code and parsing the trace.

Static analysis of the Source Code

Knowledge of the programming language allows the tool to ignore changes in comments, changed formatting, conditional compilation, changed position of programming language objects in file or objects moved to another files.

The method described above also allows detecting renamed methods, classes and other objects. These changes are not displayed on the output also. Static analysis of the source code is used for their detection.

The last category that is not displayed are declarations of variables, if they were without initializers (no default value to the variables was assigned).

Analysis of the Trace

The tool can reduce the amount of changes to methods (and other items) that were executed by the benchmark. The tool can use trace from one version, but better reduction can be done, when traces from both versions are available.

6.3. Better Presentation of the Diff

The tool tries to maximize lucidity of the changes by the structural organization of the code and syntax highlighting.

Structural Organization of the Source Code

The tool displays the source code in a structural way. List of all projects where some modification was detected is on the top level.

When user clicks on some record, all classes and structures, where some modification was detected, are displayed. Changes that are not in any class or structure are grouped in a virtual global class.

When user clicks on some class, all method with bodies and other items in the class where some change was detected are presented with the diff.

Presentation of the Diff

If there is some change in the file, not the whole file is displayed as ordinary text diff are displaying, but only the part of the file. The part also contains the context of the source code, where the change is located. It means for example, if the change was located in some method, the whole method is displayed. If the change was located in the head of the class, several lines before and behind the class head are displayed.

The tool uses highlighting (described in Chapter 5.2.1) of the source code to ease developer orientation. The tool highlights comments, strings, preprocessor directives and conditional compilation.

7. Implementation

ContextDiff is implemented in C# 2.0 language and recognizes the same language.

ContextDiff is using MCS for parsing the C# source code. MCS is C# compilator written in C# 1.0 language included in Mono [8]. Details about using MCS are described in Chapter 7.3.

ContextDiff is able to run under MS Windows with .Net Framework 2.0 or later versions and under Unix like operating systems with Mono 1.2.4 or later versions.

ContextDiff can work with substandard organization of source code files of Mono project. Source code files in Mono are divided into three groups.

Source code files in the first group are referenced from MS Visual Studio project files.

Source code files in the second group are referenced from '.sources' file. The '.sources' files contain relative paths to source code files ('.cs' files). Some '.sources' files (all '.sources' files that contains '_test' string in their names) in Mono do not contain relative paths to source code files from directory of the '.sources' file but relative paths to 'Test' directory that is located in the directory of the '.sources' file.

Example: '.sources' file 'System.ServiceProcess_test.dll.sources' contains the relative path 'System.ServiceProcess/ServiceControllerTest.cs', but source code file 'ServiceControllerTest.cs' is not located in 'System.ServiceProcess' directory but in 'Test/System.ServiceProcess' directory.

To resolve this problem *-replaceSourceString* argument was added to command line arguments (see Chapter 7.1.2).

Files in the last group are not referenced. These files are grouped by directories. The first and the second group of source code files are not disjoint.

In this chapter, usage of the tool, architecture of the tool and technical aspect of MCS are described.

7.1. User's Guide

This chapter describes the usage of ContextDiff including installation and examples of output.

7.1.1. Installation

Installation of the ContextDiff is prepared for Unix like operating systems and for MS Windows.

Unix

File 'ContextDiff.tgz' contains source code of ContextDiff with Makefile for compiling the tool. For installation, perform the following steps:

- Extract ContextDiff by entering “tar -zxf ContextDiff.tgz” command in the directory that contains 'ContextDiff.tgz' file. The command extracts the source code to 'ContextDiff' directory.
- Enter ContextDiff directory and type command 'make'. The command compiles ContextDiff and creates 'bin' directory in actual directory. All files necessary for execution of ContextDiff (ContextDiff.exe, mcs.exe and css.txt) are copied into the 'bin' directory.
- See Chapter 7.1.2 for the instructions about the execution of ContextDiff

Windows

File 'ContextDiffSetup.msi' contains an installation shield for the ContextDiff tool. For installation, perform following steps:

- Execute 'ContextDiffSetup.msi' file.
- Click the 'Next' button.
- Set the installation parameters (destination folder and tool users). Figure 7.1 describes the default parameters.

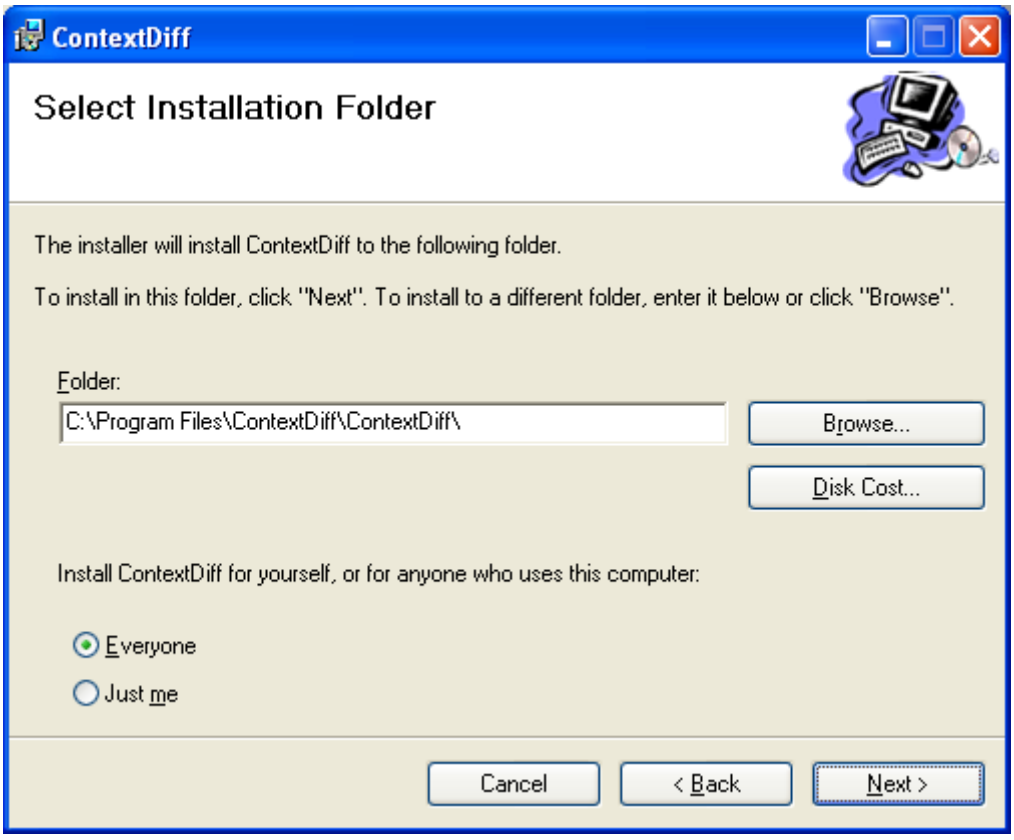


Figure 7.1: Setting destination directory and users of the application.

- Confirm installation with the 'Next' button.
- Click the 'Close' button after installation is done. All the files necessary for execution of ContextDiff (ContextDiff.exe, mcs.exe and css.txt) are copied into the installation folder.
- See Chapter 7.1.2 for the instructions about execution of ContextDiff

7.1.2. Execution

ContextDiff is a console application. All options are passed to ContextDiff by means of the command line arguments. All the valid arguments and the examples of execution are described in this chapter.

ContextDiff contains these files:

- ContextDiff.exe – the file executes ContextDiff
- mcs.exe – Modified MCS from Mono project. It is used by ContextDiff.exe for parsing source code.
- css.txt – Contains CSS (Cascading Style Sheets) classes that are included into HTML pages containing diffs created by ConetxtDiff.exe.

Command Line Arguments

All settings are passed be means of the command line arguments. All command line arguments are described in Table 7.1.

Argument	*Description*
-projectsRootDirectory1:DIR	Defines the root directory of source code of all the projects of older version of the compared application.
-projectsRootDirectory2:DIR	Defines the root directory of source code of all the projects of newer version of the compared application.
-project1:FILE	Visual Studio project file (.csproj) of the older version of the compared project.
-project2:FILE	Visual Studio project file (.csproj) of the newer version of the compared project.
-config1:CONFIG	Configuration in older version of project file that was used during the application compilation. Preprocessor directives are taken from this configuration. The argument can be used only with *-project1* argument.
-config2:CONFIG	Configuration in newer version of project file that was used during the application compilation. Preprocessor directives are taken from this configuration. The argument can be used only with *-project2* argument.
-files1:FILE1,FILE2	List of source code files of older version of the compared application.
-files2:FILE1,FILE2	List of source code files of newer version of the compared application.
-dirs1:DIR1,DIR2	List of directories that contain source code files of older version of the compared application.
-dirs2:DIR1,DIR2	List of directories that contain source code files of newer version of the compared application.
-filesDirs:DIR1,DIR2	List of directories where one file is parsed as one whole project. Path is relative to both parameter of *-projectsRottDirectory1* and parameter of *-projectsRootDirecoty2* argument. The argument can be used only with *-projectsRootDirectory1* and - *projectsRootDirectory2* arguments.
-defines1:DEFINE1,DEFINE2	List of preprocessor directives that were used during compilation of the older version of the benchmark.
-defines2:DEFINE1,DEFINE2	List of preprocessor directives that were used during compilation of the newer version of the

Argument	*Description*
	benchmark.
-display:[Console:HTML]	Type of displaying modifications detected. Console option is for displaying modifications in console. HTML option is for displaying modifications in HTML pages (must be also set *-outputHTMLDir* argument).
-outputHTMLDir:Dir	Directory where output HTML pages are created. The argument can be only used with *-display* argument set to HTML.
-minsimilarity:NUMBER	Real number between 0 and 1. The number defines how similar must two objects (classes, structures, methods etc.; not statements and expressions) be to be able to consider them as equal (see 6.1.1 and 6.1.2).
-replaceChar:Character	Character replaced from VS project and '.sources' files with path separator valid for current platform (for example: On Linux replaces '\' with '/')
-replaceSourceString: FROM,TO	For each '.sources' file that's name contains FROM string, TO string is added to the beginning of relative paths of all '.cs' files contained in this '.sources' files. The argument was added to ContextDiff due to the substandard organization of the source code files in Mono project.
-trace1:FILEPATH	Path to the trace output of benchmark older version.
-trace2:FILEPATH	Path to trace output of newer benchmark version.
--help	Displays information about valid command line arguments.
--about	Displays information about ContextDiff.

Table 7.1: Command Line Arguments.

ContextDiff can load the projects in two modes.

In the first mode, source code of only two projects (a project here means one executable file (without libraries) or one library) are compared. Projects can be defined by a list of files (*-files1* or *-files2*), a list of directories that contains source code files (*-dirs1* or *-dirs2*) or by Visual Studio project files (*-project1* or *-project2*).

In the second mode, more projects are compared (*-projectsRootDirectory1* or *-projectsRootDirectory2*). In this mode, all subdirectories are browsed and searched by all project files, *sources files* (files with extension ".sources" that contains only relative paths to source code files) and simple source code files.

Source code files that are not contained in any Visual Studio project files or any sources file are compared within the last stage. The whole directory with source code file is considered to be a project except directories defined in *-filesDirs1* or *-filesDirs2* arguments.

Example

In this Chapter, example of command line arguments for ContextDiff is given.

In the sample ContextDiff compares source code of the two versions (20070810 and 200708110) of Mono project under Linux operating system. ContextDiff is executed under Mono. ContextDiff compares only methods, properties etc. that was executed during Rijndael benchmark.

Here the example is:

"mono /ContextDiff/ContextDiff.exe
-projectsRootDirectory1:mono/mono-20070810
-projectsRootDirectory2:mono/mono-20070811
-outputHTMLDir:html/rijndael/20070810_20070811
-display:HTML
-replaceChar:
-trace1:traces/20070810/rijndael/trace
-trace2:traces/20070811/rijndael/trace
-defines1:NET_1_1
-defines2:NET_1_1
-filesDirs:mcs/tests, mcs/errors, mono/tests/cas/demand, mono/mini, mono/tests, mono/benchmark, mcs/class/Mono.Cairo/Samples/gtk, mono/tests/cas/threads, mono/tests/cas/linkdemand, mcs/class/Mono.Cairo/Samples/png, mono/tests/cas/inheritance, mcs/class/Mono.Cairo/Samples/x11, mono/tests/cas/assembly, mcs/class/System.XML/System.Xml.Serialization/standalone_tests"

The first two arguments define the directories, where the both versions' source code is stored.

Next arguments defines destination output directory.

Next argument sets the output to HTML type.

Paths in project files in Mono project source code are defined according to MS Windows style (directory separator is backslash). Next argument notice ContextDiff about this fact.

Next two arguments define the trace output files of Rijndael benchmark executed on these Mono versions.

Next two arguments define preprocessor directives which were used to compile both versions of Mono on which the benchmarks were executed.

The last argument defines directories, where each file is parsed as whole project, if it is not a part of any project.

7.1.3. Output

ContextDiff supports two types of rendering the detected modifications. The first type is output directly into console, while the second one is output to HTML pages.

Console

Rendering modifications to console is well-arranged, but reader does not see source code.

Console output can be used as an input for any third party application that can work with source code (for example some third party browser of source code that can display modifications detected by ContextDiff).

Figure 7.2: Rendering modifications to console.

The example of modifications rendered to the console is shown in Figure 7.2.

Each detected modification stars with type of modification in source code ('Created' for created statements, 'Deleted' for deleted statements, 'Modified From ' or 'Modified To' for modified statement). Modified statements are split into two

parts. The first part represents a statement before modification ('Modified From') and the second part represents the statement after modification ('Modified To').

Character '>' is rendered after the type of modification and then the type of statement as it is represented in MCS is rendered.

The next lines provide the information available. This information always starts with one space character (for the better orientation in modifications).

- Location: Location of modified object.
- Container name: Name of the class or structure, where the modification is located.
- Container member: Name of the method, constructor, destructor, property or indexer, where the modification is located.
- Member location: Start location of the container member (for example the head of the method).
- Member end location: End location of container member (for example, the last row of method).

HTML Pages

Rendering modification to HTML pages allows the user to see the detected modifications right in source code.

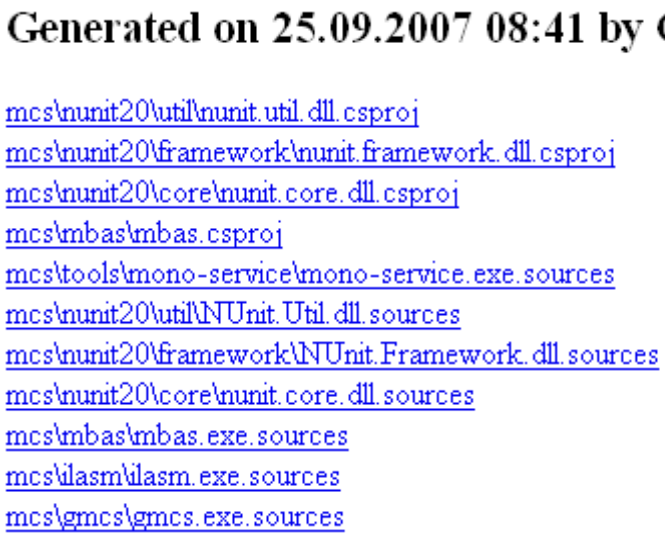


Figure 7.3: The list of modified projects.

Figure 7.3 Displays the list of the projects, where was detected some modification that was not reduced (see Chapter 6.2).

When the user clicks on some project, the list of classes and structures is shown.

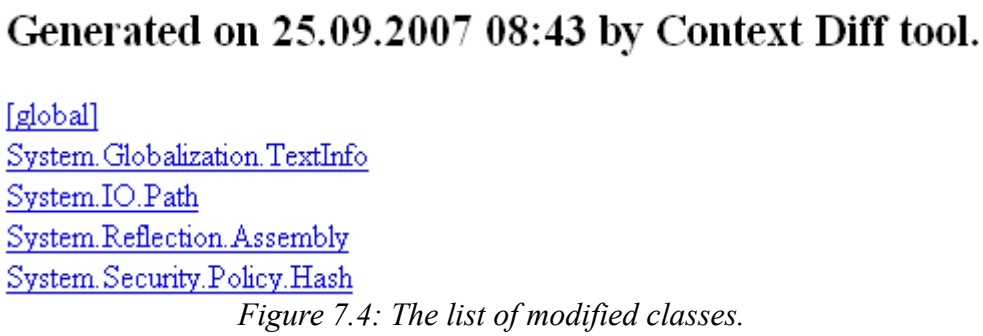


Figure 7.4: The list of modified classes.

In Figure 7.4, the list of classes and structures where some modification was detected is displayed. The word "[global]"is displayed as the first link. It is not the real class. This section contains all the changes that are not located in any class or structure (for example enumerators can be inside the class but can be also outside of any class).

When the user clicks on some class or structure, modifications in this class or structure are shown.

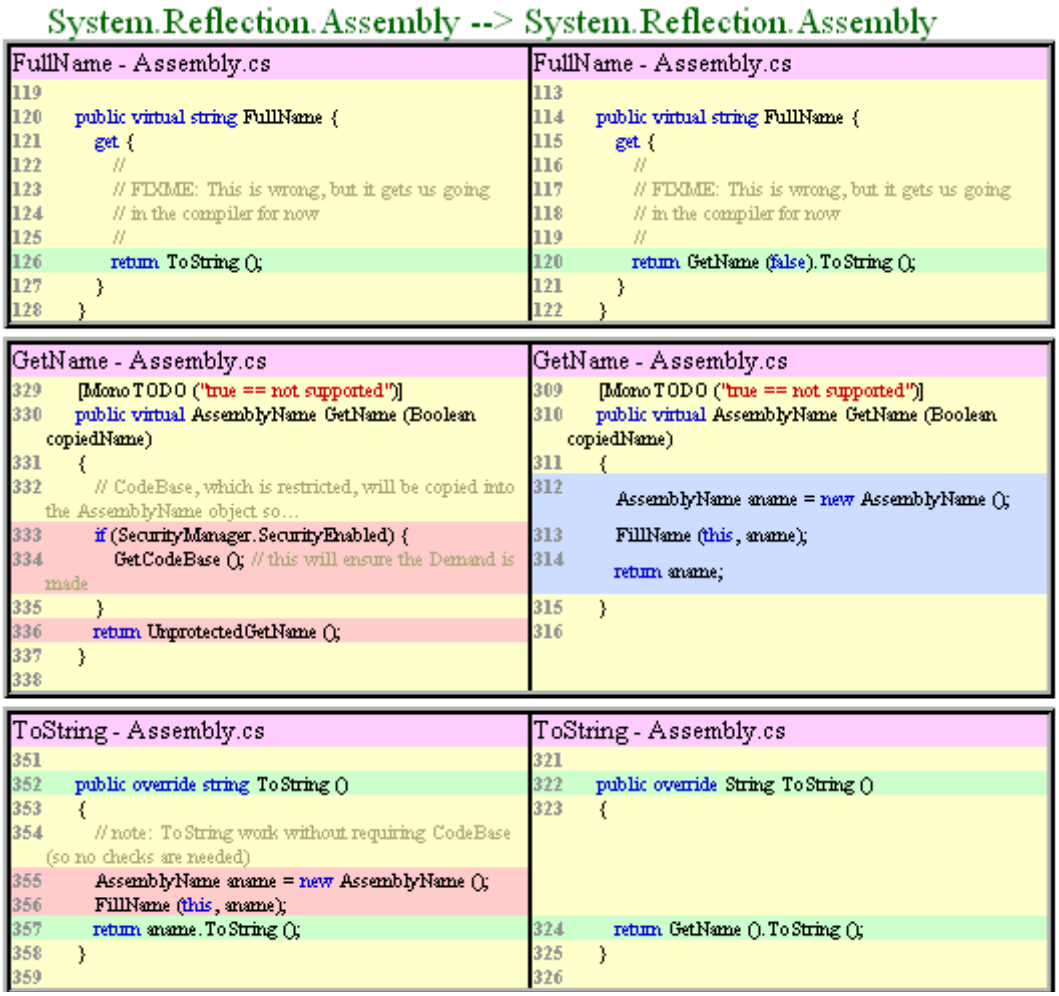


Figure 7.5: Presentation of diff for class with name Assembly.

A sample comparison of two versions of a C# class *Assembly* is shown in Figure 7.5. The older version of the class is shown in the left column and the newer one is shown in the right column. The name of the class in both the older version and the newer version is displayed on the top of the diff.

In the remaining part of the diff some methods are displayed. Only methods where some modification that can cause performance changes is detected, and which were called during the benchmark, are displayed. The names of the methods and files where the methods are located are displayed on the top of every method. The name of the method and file are displayed for both older and newer versions, because the method can be renamed or moved to another file.

Green colour represents modified statements. Red colour represents deleted statements, and blue colour represents inserted statements.

Some statements can be rendered with a light green background colour (it is not shown in this example). In the current implementation, the light green is used only for invocations of the renamed methods. From static analysis it is always not

singe valued, which method is called from this or that location. Therefore, when any definition of the method is renamed from name A to name B, and the method in some invocation is also renamed from name A to name B, it can be the same method, but it can be also the changed invocation to another method that is not connected with the renamed method. From the static analysis, it is not clear, whether it is the first option (then it should be not marked as a modification because it does not influence the functionality or performance) or if it is the second option (then it is the modification that can rapidly change the functionality or performance).

A number of the row in the file where the statement is located is on the left side of each method. Corresponding unmodified and modified statements are rendered on the same row. If it was necessary, between statements empty lines are rendered to perform alignment. Therefore there is some empty space between lines 323 and 324 in the newer version of the ToString method.

7.2. ContextDiff Architecture

The algorithm used is general but implementation is specific for the C# language. Therefore the terms that are specific for C# language are also used in this chapter. For the explanation see MSDN [20].

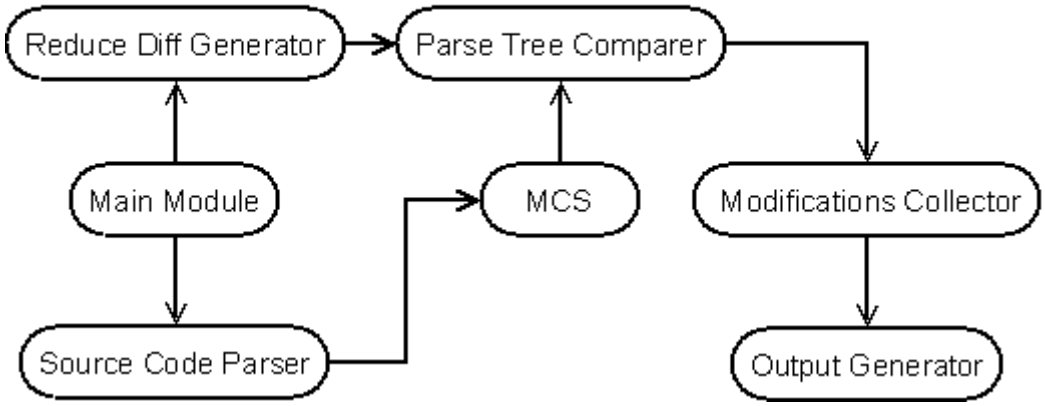


Figure 7.6: Data flow in ContextDiff

Figure 7.6 shows the data flow between the modules in ContextDiff. Detailed description of each module is given in sub-chapters of this chapter. Here are brief characteristics of the modules.

Main Module reads command line arguments from the user. The first the module pass the names of the trace outputs of both benchmarks (if available) to *Reduce Diff Generator* that reads and parse the trace outputs. Then *Main Module* executes diff for each project (source of the library or the executable file).

Source Code Parser initializes the parsing of one project. The list of the

source code files is passed to *MCS*. Two parse trees (one for each version of the project) are created. The trees are compared by *Parse Tree Comparer*. For each method, property, constructor or field *Parse Tree Comparer* is asking *Reduce Diff Generator*, if the item was used during the execution of the benchmark.

Each detected modification is stored in *Modifications Collector*. All modifications are rendered to output (Console or HTML pages) by *Output Generator* after comparison of whole parse trees.

7.2.1. Main Module

The running of the tool is controlled by several classes.

Class Main

Class *Main* is responsible for the execution of each step of the comparison. The class starts parsing arguments of the tool, reading trace files, comparison of the source code for each project and displaying results of the comparison (all parts will be described later).

Classes Arguments, ParameterException and Settings

When the tool starts, the arguments passed by means of command line are parsed by the class *Arguments* in the first step.

If any argument is not recognized, has invalid value or some arguments are in conflict, *ParameterException* exception is thrown. The exception aborts the execution of the tool and writes a message on the console.

The values read by the class *Arguments* are stored in class *Settings*. The values affect the execution of the application.

If projects are defined by the list of files, by the list of directories or by the project file, the list of source code file is generated (from the list of files, or by finding all source code files in the set of directories or from the project file).

The class *Arguments* can also write information about the application and information about arguments to console.

Class Files

If there is not only one pair of the projects compared, but the whole list of projects, all source code file names from the root directories of both list of projects are stored into the class *Files*. The class controls, which source code files have already been read. In the last step of the projects comparison all directories in the root directory of both projects lists are browsed. Corresponding directories are compared as projects, but only the source code files that have not been yet read (are not marked as red in the class Files) are read.

Class Statistics

Class *Statistics* collects statistics during the execution of the application. The statistics are written into text file in the directory for the HTML output (if HTML output is enabled) for each read project and global for all the projects.

The statistics collected are:

- amount of read files for both older and newer version of the application compared.
- amount of files rendered with some modifications to HTML pages for both older and newer version of the application compared.
- modified, added and deleted statements (collected when some modification is detected)
- modified, added and deleted lines (collected when the modifications are rendered to HTML pages; as it has been indicated above, always the whole line with a modification is always rendered with a different colour; each line rendered with this different colour (red, green or blue) is counted).

7.2.2. Source Code Parser

It is necessary to read and parse a source code before it is compared.

Class *Project* is responsible for getting the list of source code files of one project and passing this list to MCS compiler. The list can be taken from MS Visual Studio project files, sources files (these files contains just the list of source code files for the project), generated from the list of files of defined directory or can be passed as a parameter.

If preprocessor directives are not defined (by *-defines1* or *-defines2* command line argument), class *Project* reads preprocessor directives from MS Visual Studio project files (if available).

Class *Parser* starts parsing all the source code files provided by the class *Project*. MCS compiler (see Chapter 7.3) is used for parsing itself.

7.2.3. Parse Trees Comparer

The algorithm for comparing source code is described in chapter 6.1. Implementation details are in this chapter.

Each construction in C# language is represented in some class of MCS. To compare the two constructions of the source code, ContextDiff has to compare their representations in MCS classes.

Attributable Comparers

Classes that compare *attributable* objects are described in Table 7.2.

Attributable objects are those having the attributes applied. In .Net 2.0, the attributable objects are classes, structures, methods, constructors, destructors, fields, properties, indexers, definitions of constants, enumerations, members of enumerations, interfaces, delegates and events).

Comparer	*MCS*	*Description*
AttributablesComparer	class Attributable – base class for representing all attributable objects	Compares only attributes for all the attributable objects.
ConstructorsComparer	class Constructor – represents constructors	Compares lists of constructors in two classes or structures.
ConstsComparer	class Const – represents the constants definitions	Compares the lists of all the constants definitions inside the two classes or structures.
DeclSpacesComparer	class DeclSpace – base class for representing the structures, classes, enumerations and interfaces	Compares namespaces and type parameters of the two instances of DeclSpace.
DelegatesComparer	class Delegate – represents delegates	Compares the lists of the delegates inside the two classes, structures or directly in namespaces (outside any class or structure).
EnumMembersComparer	class EnumMembers – represents enumeration members	Compares the member lists of the two enumerations.
EnumsComparer	class Enum – represents enumerations	Compares enumeration lists inside the two classes, structures or directly in the namespaces (outside any class or structure).
EventsComparer	class Event – represents events	Compares the lists of events inside the two classes or structures.
FieldBasesComparer	class FieldBase – base class for representations of fields and constants	Compares the common attributes of lists of fields and constants inside the two classes or structures.

Comparer	*MCS*	*Description*
MemberCoresComparer	class MemberCore – base representation of members	Compares the lists of classes or structures inside the two classes, structures or directly in the namespaces (outside any class or structure).
MethodCoredComparer	class MethodCore – base class for representation of methods, operators, constructors and destructors	Compares the lists of methods, operators and destructors inside the two classes or structures.
PropertyBaseComparer	class PropertyBase – base class for representation of properties and indexers	Compares the lists of properties and indexers inside the two classes or structures.
TypeContainersComparer	class TypeContainer – base class for representing the structures, classes and interfaces	Compares the two classes or structures.

Table 7.2: Comparison of Attributable Objects

Statement Comparers

In MCS there are many classes representing different kinds of statements. These classes are not as complex as the classes representing attributable objects. Therefore comparing these classes is implemented only in three classes (*StatementsComparer*, *BlocksComparer* and *ExceptionStatementsComparer*).

Class *StatementsComparer* is able to compare almost all kinds of statements (except “try-catch”, “using” and “lock” statements). In this class LCS algorithm described in Chapter 6.1.3 is also implemented.

ExceptionStatementsComparer is able to compare three kinds of statements (“try-catch”, “using” and “lock”). The comparison of these kinds of statements is not included directly into the *StatementsComparer* class because their representations in MCS have the common parent class - *ExceptionStatement*. Representations of all other statements are inherited directly from the *Statement* class.

One special kind of statement called *Block* exists in MCS. The block is defined as a sorted list of statements. Some of these statements can be also recursively other blocks. The first use of the block is the representation of the method, constructor, destructor and set and get accessor bodies in properties and indexers. The second use is to group statements (for example in conditions or loops).

Class *BlocksComparer* is designed to work with the first use of the block

(the second use are blocks that are compared directly in class *StatementsComparer*). During the comparison of the two classes or structures, class *BlocksComparer* stores all the pairs of corresponding blocks. In the last step of the comparison of the classes and structures, all these pairs of blocks are compared.

Expression Comparers

The last group of constructions of the C# language is the group containing *expressions*. Expressions are parts of the statements or parts of other expressions.

Class *ConstantsComparer* compares all kinds of constants used in the code but not definitions of the constants (they are attributable objects compared by *ConstsComparer*). MCS has one class for each representation of the constant (for example signed int, unsigned int, signed long, unsigned long, string , character, ..).

Class *ExpressionStatementsComparer* compares expressions that can appear on statements (invocations, object creations, assignments and post/pre increment and decrement). These expressions can be used as normal expressions (can be parts of other expressions or some statement) but can also be a single statement stored in *StatementExpression* class. *StatementExpression* class has only one property, where this expression is stored.

The *FullNamedExpressionsComparer* class compares namespaces and types (the types in MCS are represented by: *ComposedCast* – represents the types during the cast, *TypeLookUpExpression* – represents the types with fully qualified names, *TypeExpression* – represents the types that have already been evaluated to the type, *UnboundTypeExpression* – represents unbound generic types. There are other representations, but they are created in later phase of the compilation process that is not executed by ContextDiff).

The *VariableReferencesComparer* class compares the references to the local variables and to the parameter reference.

Class *ExpressionsComparer* is called for comparing any pair of expressions. The class the first compares the types of the expressions. If they are equal, it compares the very expressions or calls another class that can compare the related type of expression (*ConstantsComparer*, *ExpressionStatementsComparer* or *FullNamedExpressionsComparer*).

Other Types

In MCS several types of objects that do not belong to any of the previous groups are defined.

The arguments of the called methods represented by the *Arguments* class are the first such type. The *ArgumentsComparer* class can compare the arguments lists of the two methods calls. Expressions (not values) representing arguments are compared.

Fixed declarators are the next such type. In MCS the fixed declarators are

represented by *LocalInfo* class, the two lists of fixed declarators can be compared by *FixedDeclaratorsComparer*.

Identifiers are represented by .Net *string* type. The *IdentifiersComparer* class can compare two identifiers. Separation the comparison of identifiers into a new class allows the implementation of the renamed identifiers detection in future versions.

Namespace entries are represented by MCS *NamespaceEntry* class. Two namespace entries can be compared by *NamespacesComparer* class.

Parameters are represented by MCS *Parameters* class and the lists of parameters of the two methods can be compared by *ParametersComparer* class.

Constraints are represented by MCS *SpecialConstraint* enumeration if they are constructors, reference type or value, or by some of the expression class in other cases. Two lists of the constraints can be compared by class *ConstraintsComparer*.

Type parameters are represented by MCS *TypeParameters* class, and the two lists of type parameters can be compared by TypeParametersComparer class.

Type arguments are represented by MCS *TypeArguments* class and the two lists of type arguments can be compared by *TypeArgumentsComparer* class.

In ContextDiff there are several other classes helping to compare the source code:

The *ArrayListComparer* class helps other classes to compare the two sorted or two unsorted lists. All the methods have to be called with one argument, which is an instance of class implementing .Net interface *IComparer*, which is used to compare pairs of items in the lists.

Information about the pair of classes or structures and the pair of objects inside the class or structure (method, property, constructor ...) currently compared is stored in *CompareContext* class.

The *NamesComparer* class is designed to store objects that are renamed in compared versions of the source code. The current version can store only the renamed methods. This class is used during comparing the invocations.

The *IdentityList* class is used to store pairs (each pair is stored in class *Identity*) of classes, structures and methods. The class is used to sort pairs according to similarity criteria (see chapter 6.1) and to choose the most similar pairs. These pairs are then declared to be corresponding pairs between the compared versions of the source code.

7.2.4. Reduce Diff Generator

Class *Trace* parses on the start of the application trace outputs of both benchmarks if they are available. If any record of the trace have invalid format (older versions of Mono sometimes generate the trace output with bugs), the

record is ignored.

Each unique record is stored in the instance of *TraceItem* class. The parameters of the called methods are stored in the *TraceParameter* class instances.

Before comparison of method, constructors and property bodies, *Trace* class is used to check, if the method, constructor or appropriate property accessor (get or set) was executed during the benchmark. If not, the bodies are not compared.

Trace outputs do not contain the access to the fields. But fields can contain the initializer, which is an executable code that can change performance or functionality. Therefore access to initializers has to be known otherwise all modified, added or deleted initializers have to be added to the list of detected changes. Initializers are called before the execution of the objects constructors (initializers of static fields are executed just once and initializers of non static fields are executed only before the non static constructors). The field initializers on the class were executed just when constructor (any for static filed; non static for non static field) was executed.

7.2.5. Modifications Collector

All detected modifications (modified, deleted and inserted items) in the source code are stored in *ModificationList* class.

The *ModificationList* class contains the list of the *Modification* class instances. One instance of the *Modification* class represents one detected modification in the source code.

The *Modification* class contains two instances of the *DisplayItem* class. One instance contains information about the modification in the older version of the source code and the second instance contains information about the modification in newer version of the source code.

The information in class *DisplayItem* consists of a modified item (statement, method, class etc.); the name of the class or structure where the modification is located; the name of the method, property, constructor etc. where the modification is located, and the display mask. Display mask is used while rendering the output to HTML, to highlight modification with colour that is not standard for type of modification (green for modified items, red for deleted items and blue for inserted items). In current implementation, display mask is used only for highlight invocations of the renamed methods with light green colour (see Chapter 7.1.3).

For deleted and inserted items, both instances of the *DisplayItem* class in the *Modification* class are defined but one of them (for deleted items - the second one, for inserted item - the first one) contains the missing item location (derived from the location of the item in the second version) instead of the item.

The *ModificationList* class can sort modifications. Firstly, modifications are sorted by the name of the class or structure. Secondly, the modifications are sorted by the name of the method, constructor, destructor, property or indexer, where they are located. If both are equal (or are not defined), modifications are sorted by

the location in source code. Items are sorted by the location of the items in the older version of the source code.

The *ModificationList* class is able to group modifications by class or structure, where they are located, or by method (or constructor, property etc.), where they are located. This is used to group modifications into methods and classes (or structures) in HTML output.

7.2.6. Output Generator

ContextDiff supports rendering the detected modifications to HTML page or directly to console (Chapter 7.1.3).

Console Output

For console output the *ConsoleOutupt* class is used. It is a simple class enumerating through modifications and rendering them to the console as described in Chapter 7.1.3.

HTML Output

The *HtmlOutput* class is used to render modifications into HTML pages. *HtmlOutput* class renders all the modifications of one project and creates index file '*index.htm'* to the directories with these modifications.

Modifications are located in the source code files and appropriate parts of these files are rendered do HTML pages. The MCS created structures are not used for rendering, because these structures do not contain the original formatting and comments. Therefore the *HtmlOutput* class also contains a simple C# parser that can recognize keywords, string constants, comments, preprocessor directives and conditional compilation in source code, and which is used for syntax highlighting.

The *ShiftList* class is used in HTML output for ensuring, that modified line will be rendered on one row (newer and older version of the modified item will be rendered on the same line). In the *ShiftList* class the list of lines where an empty space should be created in HTML output and how long the empty space should be, is stored

The list is created during adding the modifications to the *ModificationList* class while comparing the method, constructor, destructor, property or indexer bodies. The list is created here because here are computed the lines for any changes due to another purpose (storing lines in the instances of the *DisplayItem* class for the created or deleted items (not modified) as described above) and is not need to compute it again during rendering the HTML output.

The *ColoredShiftList* class is used in HTML output for correct highlighting of the modifications that are longer than one row (for example, the modified condition in “if” statement that is 3 lines long). In the *ColoredShiftList* class there is the list of the statements, where the modified items are longer than one row, and length of these statements.

The list is created during adding modifications to the *ModificationList* class while comparing the method, constructor, destructor, property or indexer bodies. The list is created here, because many representations of the source code items in MCS have not included the end location of the item (only start location). The location is evaluated from the location of the next item that is easy to access during comparing the source code but not during rendering HTML page.

The *MainIndex* class is used to create common index file 'index.htm' that has references to index files 'index.htm' of all projects. Each project output is stored in its own directory.

7.3. Mono C# Compiler (MCS)

For parsing C# source code Mono C# Compiler (MCS) is used.

MCS was chosen for the two reasons:

- It is open source software.
- The MCS development for implementing the new versions of C# language is still in progress (in the current version of ContextDiff the version where C# 2.0 is fully implemented is used). In future work, comparing the two versions of the source code written in the newer C# language versions will be easier to implement.

Modifications

MSC have to be modified for using in ContextDiff. MCS was needed to be modified for two reasons.

The first reason is that MCS was not intended to be used as a library but only as an executable file. Therefore some properties and fields that ContextDiff needs to access are not accessible (they are private or internal). Resolving this problem has two solutions. The first solution is using .Net reflection to access these properties and the field. The second solution is to modify the MCS source code. The second solution was chosen because the modifications are simple, MCS is then easier to use and the execution is faster than with usage of reflection.

The second modification was made because MCS is able to load only one source code version (only one project). MCS was modified to have two different projects loaded at the same time and switch between them.

Limitations

Using MCS brings some limitations.

The first limitation is that MCS does not contain the end location of many items (statements (except block), expressions and so on). This is the problem for the syntax highlighting of the modified items. The end location is in ContextDiff evaluated from location of the next item, but this evaluation may not be exact. Therefore the whole line with the modification is always highlighted, and when

the modification is longer than one row, it is always checked, if any of the rows is not empty or if whole row is not commented (these rows are not highlighted as modifications).

The second limitation is that MCS does not contain the location information about each keyword in the statement. The problem is for example in “if” statement. In the “if” statement the location of the “else” keyword is not known. The location is in ContextDiff evaluated from the start location of branch false. The “else” keyword has to be located before the false branch (if it exists; if not, the “else” keyword does not exist as well). Therefore a special object representing the else keyword is inserted into the list of statements by the ContextDiff itself.

8. Evaluation by Case Study: Mono Regression Benchmarking Project

This chapter reviews and evaluates the achieved goals.

8.1. Diff Sizes

The goal of the thesis is to help application programmer to locate the cause of performance degradation in the source code in reasonable time. Therefore it is necessarily to show the developer the version-to-version reduced diff. Evaluation of how this goal was fit is described in this chapter.

Mono Regression Benchmarking Project (described in Chapter 2) is used for the ContextDiff evaluation. The part of the Mono Regression Benchmarking Project is a classic text diff that compares daily versions of Mono source code. ContextDiff is being executed on the same pairs of version of Mono source code as diff in Mono Regression Benchmarking Project.

For each pair of the Mono versions source code compared versions, ContextDiff is executed with the HTTP Ping, Rijndael and TCP Ping benchmarks trace output. It means that ContextDiff was executed three times for each pair of compared versions.

Table 8.1 shows some concrete sizes of diffs. Version numbers can be found in the two first columns of the table. Sizes of diffs generated by ContextDiff that is being executed with trace output of HTTP Ping, Rijndael and TCP Ping benchmarks. *none* column indicates all diff sizes detected by ContextDiff without reduction methods executed by any other diff. Sizes of diffs generated by classic text diff included in Mono Regression Benchmarking Project are rendered under *Classic Text Diff* head. Columns with the head marked *M* displays amount of modified lines, columns with the head marked *A* displays amount of added lines and columns with the head marked *D* displays amount of deleted lines.

ContextDiff works with the method bodies statements. While rendering modifications on HTML pages the whole line where modification was detected is highlighted. Classic text diff always works with lines. That's why displaying number of lines that are highlighted as modified on ContextDiff HTML output instead of detected modifications number is more objective when comparing amounts of detected changes.

Mono contains source code written in C# language and source code written in C language. ContextDiff can compare only source code written in C# language. Therefore diff sizes generated by classic text diff in Table 8.1 are reduced to modifications in source code written in C# language.

Records for Table 8.1 were chosen randomly independent on performance change during benchmarks execution. That's why records representing diff

execution that detects no change that can affect benchmark performance are added to the table also (for example record representing comparison of versions 20070402 and 20070403).

Older Version	Newer Version	ContextDiff												Classic Text Diff		
		HTTP Ping			Rijndael			TCP Ping			none					
		M	A	D	M	A	D	M	A	D	M	A	D	M	A	D
20060208	20060212	9	18	16	12	1	2	12	1	2	926	1042	524	2260	4989	2131
20061102	20061113	22	14	7	22	6	4	22	6	4	1309	1218	1044	5016	15962	1759
20070101	20070102	3	1	2	4	0	0	4	0	0	72	50	55	82	203	65
20070102	20070103	1	1	1	9	4	5	9	4	5	235	115	97	227	1320	134
20070103	20070104	3	2	2	7	3	5	7	3	5	101	29	21	5574	592	59
20070104	20070105	5	5	1	3	3	1	3	3	1	152	572	554	933	891	98
20070128	20070130	5	14	3	15	7	4	15	7	4	182	268	171	902	30743	135
20070226	20070227	1	0	0	12	4	2	12	4	2	88	135	85	447	622	449
20070302	20070303	1	2	0	11	4	3	11	4	3	120	246	131	176	1723	303
20070310	20070311	0	0	0	7	0	2	7	0	2	123	26	150	327	346	159
20070319	20070320	0	0	2	0	0	2	0	0	2	91	255	283	243	1604	318
20070402	20070403	0	0	0	0	0	0	0	0	0	61	75	65	124	218	135
20070405	20070406	1	0	0	0	0	0	0	0	0	54	19	10	100	338	48
20070628	20070629	1	1	0	1	1	0	1	1	0	68	98	3	18	275	12
20070630	20070701	2	10	1	2	0	1	2	0	1	77	13	4	66	73	35
20070812	20070814	9	12	1	8	12	1	8	12	1	111	162	86	169	1219	199
20070816	20070817	4	2	2	3	0	0	4	2	2	139	266	72	555	1303	156
20070819	20070820	0	0	0	0	0	0	0	0	0	91	92	25	56	535	25
20070917	20070918	2	4	0	2	4	0	2	4	0	154	114	86	221	1103	72
20070922	20070923	1	0	0	1	0	0	1	0	0	93	129	21	185	552	52
20070929	20070930	14	34	12	0	0	0	0	0	0	99	46	20	64	215	61

Figure 8.1: Amount of modifications in source code of Mono

Sizes of 'benchmark diffs' (diff generated by ContextDiff from code executed only by one of the three tested benchmarks) in many cases have less than one percent of sizes of diffs that are generated by the classic text diff. Cases when sizes of 'benchmark diffs' are higher than ten percent of size of diff generated by classic text diff are rare. The ratio is so small because of small benchmark code coverage.

Sizes of 'non-benchmark diffs' (diff generated by ContextDiff that is not reduced to code executed by any benchmark) are in many cases comparable to sizes of diffs generated by classic text diff. There are several reasons for the difference between these diff sizes:

- ContextDiff do not include source code from all the branches of conditional compilation. All branches cannot be included, because it can

create syntax incorrect source code, which cannot be read by the language sensitive diff.

- ContextDiff do not include added or deleted methods if they are not virtual. Adding these methods cannot affect functionality or performance (it applies to definitions of added methods only, not to their invocation).

- Script executed in Mono Regression Benchmarking Project does not include deleted source code files (but added source code files do). Deleting source code files is not as common as adding, but sometimes it also happens.

- ContextDiff include all the files of each one project. But there is one file (*Consts.cs*) that is included in more projects (from 59 to 71, depends on the concrete Mono version). This file contains constant with version of the Mono. The constant is changed in most of tested versions (only between versions 20070319 and 20070320; 20070402 and 20070403; and 20070405 and 20070406 the constant is unchanged). ContextDiff shows this modification in all projects that include the file.

- ContextDiff do not include modifications that cannot have effect on performance generally - like changed comments or formatting. Comments are in source code of Mono rare, but for example file *CodeIdentifiers.cs* have changed formatting between version 20061102 and 20061113. Line ending of the file was changed from Windows style (CR LF) to Unix style (LF). Classic text diff marks almost all lines of the file as modified. Remaining lines are marked as added.

ContextDiff proved that it is useful also without reduction of located modification by trace output.

Table 8.1 contains average rate (in percent) of modification number rendered by ContextDiff for all the 'benchmark diffs' and 'non-benchmark diff' to modifications detected by classic text diff.

HTTP Ping			*Rijndael*			*TCP Ping*			*none*		
M	*A*	*D*	*M*	*A*	*D*	*M*	*A*	*D*	*M*	*A*	*D*
2.2 %	1.6 %	1.7 %	1.8 %	0.2 %	1.1 %	1.8 %	0.2 %	1.2 %	75.5 %	18.6 %	81.0 %

Figure 8.2: Average rate of changes

The average rate of modified lines detected by 'non-benchmark diffs' to modified lines detected by classic text diff is so high (75.5%) because of 'Consts.cs' file, which is included in many projects of one version of Mono. If only unique modifications are being count the rate is 28.9%.

The average rate of deleted lines detected by 'non-benchmark diffs' to deleted lines detected by classic text diff is so high (81%) because the script in Mono Regression Benchmarking Project that compares source code by classic text

diff do not include deleted source code files.

8.2. Location of Concrete Performance Change

ContextDiff was also used to locate the concrete performance change.

The biggest performance degradation detected by Mono Regression Benchmarking Project was found during the execution of HTTP Ping benchmark when comparing performance of Mono versions 20050404 and 20050405. Performance of the benchmark on version 20050405 was about 39.29% worse than on version 20050404.

Classic text diff detected 193 modified lines, 1260 added lines and 373 deleted lines in total amount of 1,597,702 (in version 20050405) lines in the C# source code. The modifications were located in 48 '.cs' files.

ContextDiff rendered only 5 lines as modified and only 11 lines as deleted.

All deleted lines were located in class *Socket*. One deleted line represents deleted field *supportsAsync* with its initializer. Remaining deleted lines were contained in methods *BeginReceive* and *EndReceive*. By the comments in subversion where the source code of Mono is stored, the changes were made to remove unused asynchronous IO code.

Modified lines were located in classes *Char* and *String*. Two overloads of method *ToLower* and method *ToUpper* of class *Char* and two overloads of method *ToLower* of class *String* were modified. By the comments in subversion the changes are made to implement culture sensitive *ToLower* and *ToUpper* methods which behave exactly the same as MS.NET does.

By the authors of Mono Regression Benchmarking Project, it was experimentally verified if the performance degradation was caused by removing parts of the source code in class *Socket* or by implementing culture sensitive *ToLower* and *ToUpper* methods in classes *Char and String*.

To verify that some modification caused performance change new sub-versions of Mono are to be created. One version is created right before the modification and one version is created right after that. Then benchmark is executed for both these versions. If the benchmark on the first sub-version has the same performance as the benchmark on version 20050404 and the benchmark of the second sub-version has the same performance as benchmark on version 20050405, it means that the modification is the reason of performance degradation. If the benchmark has the same performance on both sub-versions, the modification cannot reason the performance change. If none of these two options occurs, the modification is the cause of performance change, but another modification could cause performance change also.

This experiment successfully found the performance degradation cause in implementation of culture sensitive *ToLower* and *ToUpper* methods.

The cause of performance degradation without using ContextDiff location

can be found by using classic text diff. The one need to choose some modification that was detected by classic text diff and confirms that the modification causes the performance degradation. If the modification was not confirmed as the cause of performance alteration, another modification is chosen ... Confirming, if one modification causes of performance change is a time consuming process. Locating the cause in all the modifications detected by classic text diff can take several days (if the change that cause performance degradation was not chosen at the beginning of search)

9. Conclusion

In large projects, finding the cause of performance degradation detected by regression benchmarking is a time consuming process.

The number of the source code modifications that the one has to investigate is too high. In this thesis two methods how to automatically reduce this number were proposed. The first method is to remove these source code modifications that cannot have effect on performance generally using static analysis and parse tree comparison. The second method is to remove changes that cannot affect concrete benchmark using off-line trace analysis.

Both methods were implemented into language sensitive diff called ContextDiff, which can compare source code of applications written is C# language. The tool proved that the methods can rapidly improve the location of the cause of performance change.

The tool was tested on real system – the sources of Mono, open source .NET implementation with 2.6mil of C# code. It reduces number of modified lines average to 1.9%, number of added lines to 0.7% and number of deleted lines to 1.3% compared to classic text diff.

The tool was written to be able to be included in evaluation of Mono Regression Benchmarking Project.

Future Work

ContextDiff still have potential to be improved to increase performance of locating causes of performance degradations by evaluating probability that some change can affect performance (methods described later). Detected modification can be then highlighted or sorted by this probability.

The first method, how the probability can be evaluated, is by the type of modification. Some types of modifications have higher probability of causing performance change than the other types.

The second method is to use profiling. Modifications in time consuming methods and modifications in methods that have altered their time severity have higher probability of causing performance change.

If ContextDiff can be useful in future projects, recognizing of new versions of C# language have to be implemented also. Current version of ContextDiff can recognize source code written in C# 2.0. Implementation of parsing C# 3.0 language is still not fully implemented in MCS, but implementation is in progress.

10. Bibliography

[1] Adam Kolawa, Regression Testing, http://www.wrox.com/WileyCDA/Section/id-291252.html, 2007

[2] Adam Kolawa, Functionality Testing, http://www.wrox.com/WileyCDA/Section/id-291048.html, 2007

[3] Free Software Foundation, Benchmarking GCC, http://gcc.gnu.org/benchmarks, 2006

[4] J. Andrews, Linux: Benchmarking 2.6, http://kerneltrap.org/node/4940/print, 2005

[5] DOC Group, TAO performance scoreboard, http://www.dre.vanderbilt.edu/stats/performance.shtml, 2006

[6] Sun Microsystems, Overview of Solaris patch system testing and performanceregression testing, http://sunsolve.sun.com/pub-cgi/show.pl?target=patches/sys-and-perf-test, 2006

[7] Tomáš Kalibera, Mono Regression Benchmarking Project, http://dsrg.mff.cuni.cz/projects/mono/, 2004-2007

[8] Miguel de Icaza, Mono, http://www.mono-project.com, 2004-2007

[9] Microsoft Corporation, The C# Language, http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx, 2007

[10] CollabNet, Subversion, http://subversion.tigris.org/, 2000

[11] Dick Grune, Concurent Versions System, http://www.nongnu.org/cvs/, 1980s

[12] Microsoft Corporation, Visual Studio 2005, http://msdn2.microsoft.com/en-us/vstudio/aa973782.aspx, 2007

[13] The Eclipse Foundation, Eclipse Homepage, http://www.eclipse.org/, 2007

[14] Sun Microsystems, Inc. and CollabNet, Inc., Net Beans Homepage, http://www.netbeans.org/, 2007

[15] Satish Chandra Gupta, Need for speed -- Eliminating performance bottlenecks, http://www.ibm.com/developerworks/rational/library/05/1004_gupta/, 2005

[16] Michael D. Ernst, Static and dynamic analysis: synergy and duality, http://people.csail.mit.edu/mernst/pubs/staticdynamic-woda2003.pdf, 2003

[17] Chris Hankin and Thomas Jensen, Security and Safety through Static Analysis, http://www.ercim.org/publication/Ercim_News/enw49/jensen.html, 2002

[18] Microsoft corp., .NET Framework Conceptual Overview, http://msdn2.microsoft.com/en-us/library/zw4w595w.aspx, 2007

[19] Francois Nicolas, Eric Rivals, Longest Common Subsequence Problem for Unoriented and Cyclic Strings, http://hal-lirmm.ccsd.cnrs.fr/docs/00/12/01/52/PDF/NR-TCS-1006.pdf, 2006

[20] Microsoft corp., MSDN, http://msdn2.microsoft.com, 2007