ISSN 0361-7688, Programming and Computer Software, 2013, Vol. 39, No. 1, pp. 49–55. © Pleiades Publishing, Ltd., 2013.
Original Russian Text © V.O. Savitskii, D.V. Sidorov, 2013, published in Programmirovanie, 2013, Vol. 39, No. 1.

Fast Analysis of Source Code in C and C++

V. O. Savitskii and D. V. Sidorov
Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: ssavitsky@ispras.ru, sidorov@ispras.ru

Received September 10, 2012

Abstract—Static analysis is a popular tool for detecting the vulnerabilities that cannot be found by means of ordinary testing. The main problem in the development of static analyzers is their low speed. Methods for accelerating such analyzers are described, which include incremental analysis, lazy analysis, and header file caching. These methods make it possible to considerably accelerate the detection of defects and to integrate the static analysis tools in the development environment. As a result, defects in a file edited in the Visual Studio development environment can be detected in 0.5 s or faster, which means that they can be practically detected after each keystroke. Therefore, critical vulnerabilities can be detected and corrected at the stage of coding.

DOI: 10.1134/S0361768813010064

INTRODUCTION

A vital issue in software development is its insufficient reliability. At the final stages of the development—debugging and testing—the software is checked against the specifications. However, the available testing methods cannot detect all the defects. For that reason, in the projects for which the absence of vulnerabilities is critical, additional defect detection techniques are used. Static analysis is one such technique.

Static analysis is a method of detecting certain types of situations or patterns that can indicate the presence of a vulnerability in the source code [1]. These types include memory leaks, null pointer dereference, buffer overflow, and cross platform bugs. A simple example is a compiler warning about an uninitialized variable. However, a compiler cannot detect complex defects, such as null pointer dereference, because of performance requirements. For that reason, static analysis is typically implemented as a separate technique.

As a result of the first component of a static analyzer, the abstract syntax tree with attributes is obtained. Simple defects can be detected at this stage. To describe the patterns that can be marked as defects in a simple and fast manner, one may use the KAST language (see [2]). This language is used in the components of the Klocwork Insight static analyzer [3]. More complex defects are sought using the control flow graph, which is obtained by transforming the syntax tree. Since there are many execution paths in the control flow, this stage of the analysis takes more time, and analyzing projects containing millions of lines of code can take several hours. For that reason, the static analysis is typically performed on a powerful server once a day for the whole project. As a result, the detection and, which is more important, correction of even simple defects can take too long.

If a programmer notices a warning about an actual bug at compile time, it is not difficult to correct it. A bug is harder to correct if it was detected by a user that has purchased a finished product. It is even more difficult to diagnose and fix a bug in embedded software. Thus, the "cost" of a defect rapidly increases with the time between the bug injection and its detection. The earlier a defect is detected, the easier it can be fixed.

Therefore, the acceleration of the static analysis is of key importance. Typically, the time needed for three stages of the source code analysis—preprocessing, parsing, and semantic analysis—is comparable with the time needed to actually spot defects. For C and C++ projects, this time can be significantly reduced by saving the results of the header files analysis for future use.

Typically, the structure of the source code file is as follows: first, the preprocessor directives for including

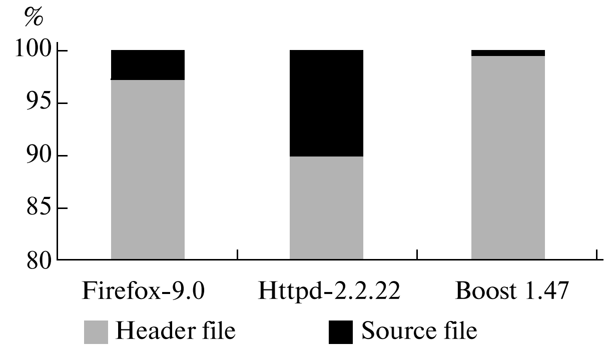


Fig. 1. Relationship between the number of lexemes in header and source files.

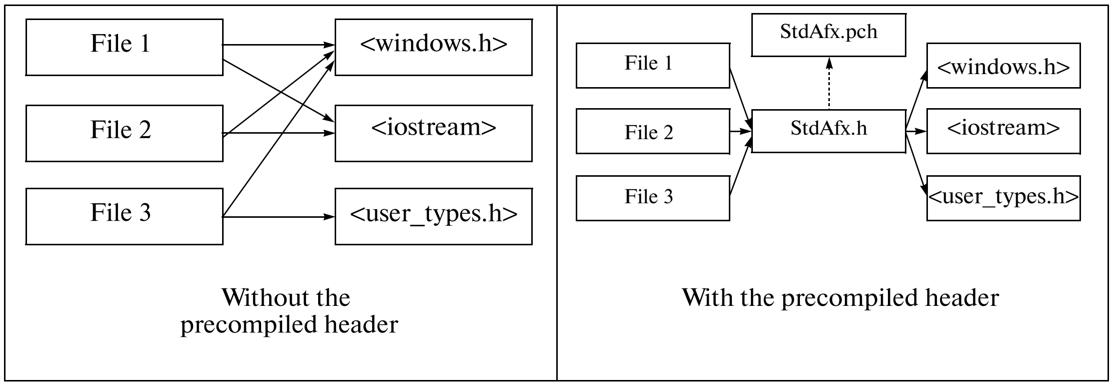


Fig. 2. The structure of using precompiled headers.

header files are written, and then go the definitions concerning the file itself. Here is an example of such a file:

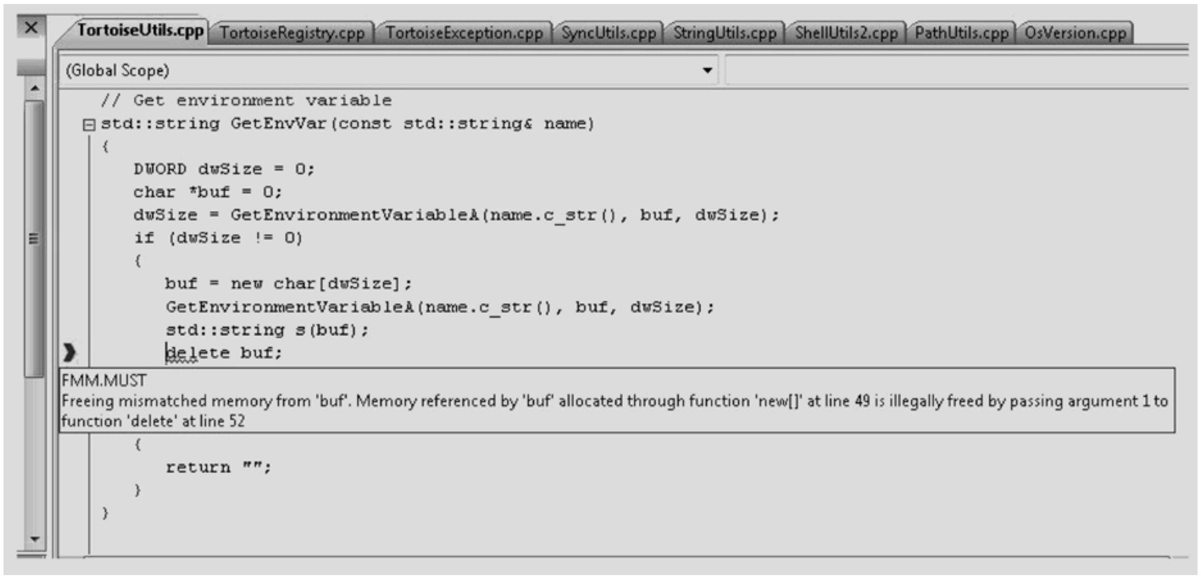
```
#include <windows.h>
#include <iostream>
#include <user_types.h>

typedef unsigned int boxes;
```

The header files usually include declarations and definitions to be used in many files. Each file can use only a small number of definitions from a header file but must include all of them. Furthermore, the same headers are analyzed many times. As a result, the compilation time of the whole project increases unreasonably. If the results produced by the preprocessor, parser, and semantic analyzer could be saved and used for the further operation of the analyzer, the total time needed to process each source file would be reduced. We consider three methods based on this observation.

The *incremental analysis* assumes that the context for multiple analysis of different versions of the same file is saved. This will allow us to accelerate the repeated analysis of the file when it is edited so that the analysis could be performed within a second, which is about the text input rate. The *lazy analysis* is based on the postponement of the analysis of unused objects in header files. Finally, the *header files cashing* method is based on saving the traces of the constructed semantic elements for reproducing them in the source files analysis. Below, we describe these methods in more detail.

INCREMENTAL ANALYSIS

Typically, a programmer looks through the results of static analysis using some interface, makes corrections, and checks the results the next day. In this case,



```
// Get environment variable
std::string GetEnvVar(const std::string& name)
{
    DWORD dwSize = 0;
    char *buf = 0;
    dwSize = GetEnvironmentVariableA(name.c_str(), buf, dwSize);
    if (dwSize != 0)
    {
        buf = new char[dwSize];
        GetEnvironmentVariableA(name.c_str(), buf, dwSize);
        std::string s(buf);
        delete buf;
```

FMM.MUST
Freeing mismatched memory from 'buf'. Memory referenced by 'buf' allocated through function 'new[]' at line 49 is illegally freed by passing argument 1 to function 'delete' at line 52

```
    {
        return "";
    }
}
```

Fig. 3. Representation of defects in the Klocwork plug-in for Visual Studio.

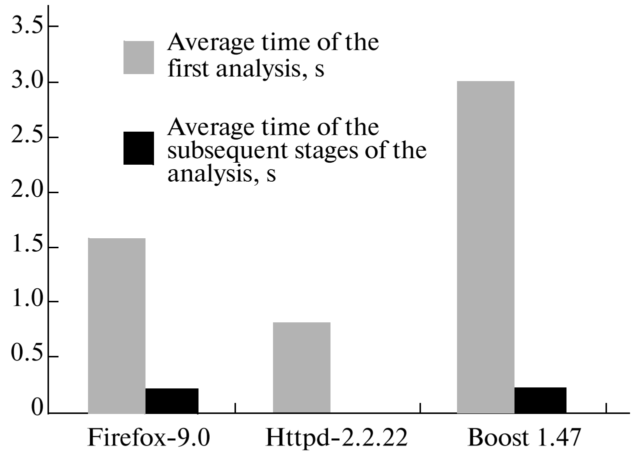


Fig. 4. Comparison of the time of first analysis with the subsequent analysis stages.

defects can get, for example, in the night build that will be used by testers. Moreover, the programmer can turn his or her attention to another task the next day, and additional time can be needed to return to yesterday's task. It is clear that the ability to reduce the time of this cycle is of great importance. For example, if the bugs are detected simultaneously with the input and editing of the program text, then the time needed to fix the defects will be minimized. The incremental analysis makes it possible. When a programmer changes the source code, the syntax tree must be rebuilt, analyzed, and the new list of defects must be compared with the preceding one. In order to rapidly build the new syntax tree, we can construct it incrementally using the results of the preceding analysis iterations. The analysis of several open-source projects shows that more than 90% of symbols processed by the compiler come from header files (see Fig. 1).

A natural approach in this case is to process the common header files only once. This technology is known as precompiled headers. For example, in the variant implemented in the Microsoft Visual Studio compiler [4], a common header is created for the entire project or its part that includes all the header files needed in different source files of the project. Then, the header include directives in each file are replaced by a single include directive for this common header (see Fig. 2).

The compiler is first given the command to create a precompiled header. The compiler creates a file in which it stores the table of preprocessor macros, the table of found identifiers, and other data needed for further parsing. When the project is built the next time, the compiler only spends time loading the saved data. This considerably reduces the time of analysis of each individual source file. A somewhat different approach is used in the gcc compiler [5]. gcc allows one to compile the header file as an ordinary file in the project. This creates a file with the extension .gch in the folder, where the header file resides. When the include directives are processed, the compiler first looks for the corresponding .gch file and uses it if possible. There is another method—frequently used headers are gathered into a single header. This combined header is included in the project files using the –**include** option. If the headers contain include guards, then the headers included in the combined header are processed only once. The method used in gcc is preferable to the method used in the Microsoft Visual Studio compiler because it does not require the source code of the project to be modified.

ON-THE-FLY ANALYSIS

The precompiled header technique can be generalized for the case of a single file that is analyzed repeatedly. For each file that is opened in the development environment, the analyzer creates the cache of the following form. The analyzer processes the input sequence of symbols keeping track of the appearance of the first lexeme from the source file itself (after the series of include directives). In the following example, this place is marked by the dashed line.

```
#include <stdio.h>
#include <stdlib.h>
---------------------------------
typedef char number;
#include <inlined.h>
```

Notice that the last header in this example is not included in the cache because the corresponding include directive is after the first lexeme of the source file; however, such situations are rarely encountered. At the time when such a lexeme is detected, the "snap" of the data processed by this time is stored in memory. It includes the preprocessor symbol table, the parser symbol table, the required semantic elements, and the state of identifier scope. The current state of operation of the analyzer components that need the information about the whole file to detect a defect is also saved. Then, the remaining part of the file is analyzed. After the user has modified the file, the defect search is started with a certain delay. First, the analyzer must be switched to the state that it had at the moment of creating the snap. For example, for the semantic analyzer, this implies that the elements that were added to these scopes during the analysis of the source file itself are deleted from the scopes coming from header files. The memory that was dynamically allocated for the structures in the main part of the source file must be also freed. To simplify the transition to the state before the analysis of the source file, two memory pools—one for all the header files and the other for the main part of the file—can be created. Then, the memory will be freed in one operation if all the references to the elements in the second pool are removed from the pool corresponding to the header files. Then, the syntax analyzer starts the analysis from the first

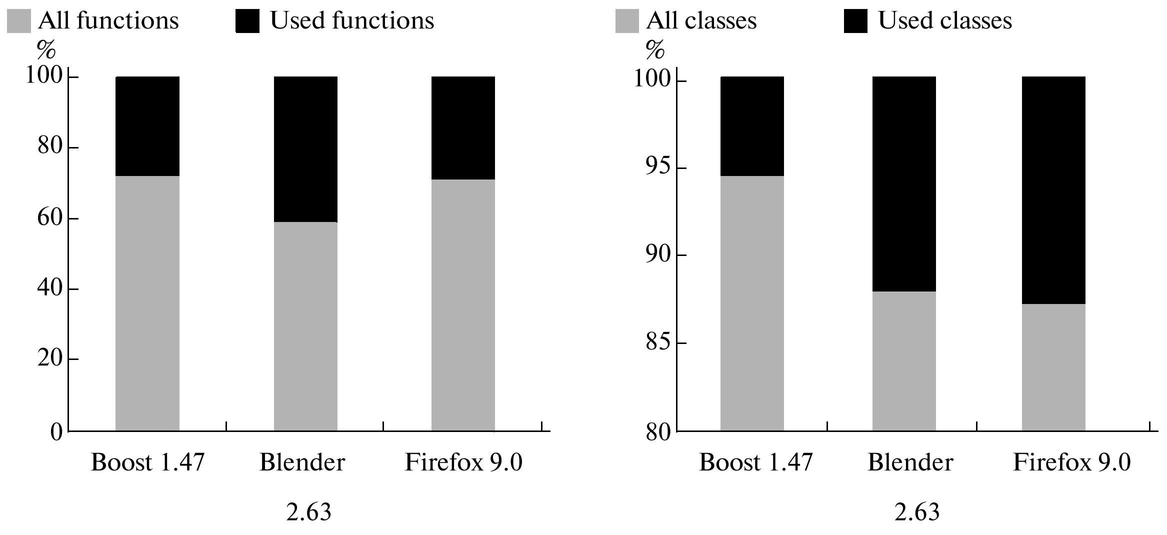


Fig. 5. Relationship of the used and all the objects.

lexeme after the header files. The syntax tree of the source file incrementally complements the saved tree of the headers. As a result, the complete syntax tree will be identical to the tree constructed in the course of the complete build.

The search for defects based on the syntax tree is a fast operation; therefore, there is no sense in restricting the search area by the part of the file (a class or a function) where modifications were made. However, this optimization is important for accelerating the construction of the intermediate representation and for the data flow analysis. To this end, as the tree is traversed, a control sum for each function is calculated on basis of its lexemes. In the next iteration, the analysis is restricted to the functions for which the control sum has changed and the dependent functions.

When defects are detected, they are represented in the window containing the source code in the form of a column with markers or are underlined directly in the code. Figure 3 demonstrates how defects are represented in the Klocwork plug-in for Visual Studio.

It is reasonable to begin the analysis with a small delay (for example, half a second) after the time when the user stops editing the text. Often, the source code is syntactically incorrect at this time. For that reason, error processing is of major importance. The syntax analyzer must correctly recover after an error is detected and stop the further search for defects; otherwise, the probability of false positives and false negatives is high. It is also necessary to check if the context has changed since the time when the snap for the header files was made. Here, the context is interpreted as a set of compiler options, contents of the header files, and the set of defects to be found. If one of these components has changed, the header files must be analyzed again.

Figure 4 illustrates the results: due to the incremental file analysis, the time of repeated analysis stages is reduced by a factor more than ten compared with the time needed for the first analysis. This makes it possible to perform the analysis after every keystroke.

“LAZY” ANALYSIS

The incremental analysis method is useful only when the same file is processed repeatedly. Furthermore, the source file should be edited in an integrated development environment. In this section, we consider a syntax analysis method that makes it possible to accelerate both the analysis of individual files and the complete project build. To reduce the time needed for the repeated analysis of header files, the unused objects can be excluded from the analysis. This approach is partly implemented in Microsoft Visual Studio [6]. The header files included in the project often contain many templates that are never instantiated. When the parser detects the header of a template, it skips its body and keeps only the context needed to parse it further. If the parser later discovers this template instantiation, its body is analyzed. Otherwise, the template body is not analyzed. As a result, the following code does not cause the compiler error:

```
template <typename T> class error_example {
>>>> example error <<<<
};
```

Such a behavior of the parser can be called “lazy” because the analysis of the template body is postponed as long as possible or is not performed at all. This method can also be extended to other basic elements of the code—bodies of functions, structures, and classes (not necessarily template-based ones).

To evaluate the number of objects that are actually used in the project files, we analyzed several opencode projects. The set of used functions and classes is determined as follows. All the objects (classes and functions) in the source code are scanned. For each class, its members and member functions are scanned. For each function, its body is scanned. If a class or a function that is not in the set of the scanned objects is encountered, it is added. Thus, the set of functions and classes that are required to compile the source file is obtained. The other functions and classes may be skipped in the analysis.

Figure 5 shows the total number of lines in the classes and functions in the projects under examination compared with the number of lines in the used classes and functions.

It turned out that almost a half of all lines in the classes and almost 90% of lines in the functions may be skipped in the analysis. Let us discuss in more detail how such an analyzer can work.

Typically, the C and C++ parsers use the bottom-up LALR(1) parser algorithm [7]. Such a parser looks through the sequence of lexemes from left to right and makes the decision about which operation (shift or reduce) to apply based on the lookahead of one lexeme. In the case of shift, the current lexeme or nonterminal is moved to the stack; in the case of reduction, the symbols corresponding to the reduction rule are extracted from the stack. C and C++ languages have ambiguous grammars, and the selection of the reduction rule often depends on the identifier type. For that reason, shift/reduce and reduce/reduce conflicts often occur in the grammar construction. Such conflicts are resolved by using the generalized LR(1) analysis method: the stack is duplicated for each feasible variant, and the analysis is performed for each stack individually until a syntax error is discovered on one of them or both stacks merge.

To add the postponed parsing functionality for classes and function bodies, a limited number of changes are needed. When the parser encounters an opening brace in the course of lexeme scanning, it must skip all the lexemes until the corresponding closing brace while maintaining the balance of braces. The parser should replace the tree node corresponding to the class body with a special node containing the information needed to parse this body later if required. This information includes the pointers to the first and the last symbols of the class body in the sequence of lexemes and the pointer to the current scope. Later, if it is required to create a class object or to access an object within the class scope, the parser will restore the corresponding scope as the current one and invoke the analysis of the class body. In turn, the analysis of the member function bodies and nested classes can also be postponed. The same is true of the bodies of ordinary functions.

When the context for the postponed class body parsing is saved (the pointers to lexemes and scope),

Table 1

Project	Ordinary mode, s	"Lazy" mode, s	Increase in the analysis speed
Boost 1.47	477	278	42%
Blender 2.63	168	132	22%
Firefox 9.0	1.44	735	36%

the exact time when the scope was stored must also be saved. This is required to avoid errors in a situation when the names that are already in the scope are added once more but with different meaning. For example, when the function foo is declared in the following example, the current scope already includes the name x of the type int. If we postpone the parsing of the function body until its call, the name x will denote a type when the scope is restored as the current one. Therefore, we must store the state of the scope along with the pointer to it. For example, we can introduce the concept of revision as in version control systems. When a name is added to a scope, the scope's revision is incremented, and the added name will store this revision number. Later, the revision number can be used to find the name in the scope. In the example under examination, the function body was saved with the scope of revision 2; therefore, all the names with a revision number greater than two were added later, and they should not be taken into account when the function body is analyzed.

```
template <int wheels> class Vehicle {};
---------------------------------------
int x; // x: rev. 1
void foo() { // foo rev. 2
  x = 2;
}
typedef int x; // x: rev. 3
foo();
```

In this example, the function body will be analyzed only when the parser encounters the call of the function foo. For this purpose, the global scope of revision 2 will be set (which was active when the function was declared). When determining which declaration at the upper level corresponds to the name x in the function body, revisions are used. The declaration of the type synonym does not fall in the list of candidates because it was added to the scope with revision 3; that is, after the function body was declared.

Since the template *Vehicle* is not used, its body is not analyzed.

The lazy analysis is useful only for the elements in header files because the source files usually use all their declarations; furthermore, the size of the source files is not large compared with the size of header files.

Table 2

Semantic action	Operation code	Arguments
Create the element “global scope”	1	Identifier of the scope
Create the element “scope”	2	Name identifier
Add identifier to the scope	3	Identifier of the scope, name identifier

RESULTS

Table 1 shows the total time of the syntactic and semantic analysis for all project files in the lazy mode.

It is seen that the gain in time is the greatest for the project with the largest amount of code in C++ (project Boost 1.47). These results also show that the dependence of the analysis time on the number of code lines is nonlinear. For example, for the declaration of a variable, a single semantic element is created, which is added to the current scope. When an array is declared, several such elements are created; when a template is described, the template structure must be saved along with the semantic elements. For that reason, the increase in the analysis speed does not correspond to the estimates based on the number of skipped code lines.

CASHING HEADER FILES

Precompiled headers provide a means for the optimization of the analysis time of header files. However, they have a significant drawback—the header inclusion context can be easily violated. For the two following files, the inclusion context is different; therefore, the same precompiled header cannot be used:

test1.cpp
```
#include "precompiled_header.h"
#include "test.h"
```

test2.cpp
```
#include "test.h"
#include "precompiled_header.h"
```

The problem is that a precompiled pch-file contains many headers, but one cannot use the saved information for only one header. The header caching method assumes that the results of analysis are saved for each individual header. Then, this information is loaded as header include directives are detected. A similar project was developed by the gcc group (see [8]), but it was not brought to a ready-to-use product.

The cache for the header file includes preprocessor symbols and semantic elements. This information is sufficient for the analysis of the source file, and there is no need in saving the syntax tree for the header. To save all the constructed semantic elements and recover the references between them in the subsequent analysis, large resources would be required. Instead, it is reasonable to save the trace of creating these elements; that is, to save the sequence of codes of the semantic actions that were performed in the course of the analysis. A record in the trace consists of the operation code and its arguments. For example, if we use the codes of operations listed in Table 2, then the sequence of actions performed in the analysis of the following code fragment is described by the trace presented in Fig. 6.

The analysis algorithm is as follows. When a header file is first discovered, the trace of semantic analysis actions and the list of the defined macros are saved for this file. The static analysis based on the syntax tree for this header file is also performed because the tree would not be saved.

Then, in the course of the analysis, each header file include directive is checked for the existence of a ready-to-use element trace. If such a trace is available, it is played (that is, all the actions are performed). As a result, the same semantic elements as in the complete analysis will be constructed, but no time is spent to do preprocessing, syntactic, and semantic analysis.

Sometimes, a header file is included in different contexts in C and C++ projects; that is, the list of the defined macros is different. In this case, the sequence of lexemes for the analysis is different, and the traces for each variant of the context must be saved. Table 3 shows the results of the analysis of some open-source projects; it is seen from these results that the number of different traces for each header is typically three or four.

Presently, the approach proposed in this paper is at the stage of implementation. Testing the current prototype on source files of various projects in C showed that the semantic analysis is accelerated by a factor of eight.

```
namespace ns;
```

1:global	2:ns	3:global:ns

Fig. 6. Structure of the internal representation of the trace instructions.

Table 3

	Different headers (DH)	Different traces (DT)	DT/DH	Average trace size, KB	Largest trace size, KB
Postgres	624	2058	3.30	42	7.289
Linux	5002	16827	3.36	48	6.394

CONCLUSIONS

The methods described in this paper are not new; however, they have never been applied to static analysis. These methods considerably reduce the project analysis time, which makes qualitative improvement in the process of software development. Certain classes of bugs can be reliably detected by the static analysis; if these bugs are detected quickly, they can be corrected at the earliest stage of the development, namely, at the stage of editing the source code. As a result, at the later stages—building, loading into the version control system, and testing—such bugs are almost excluded. Thus, the static analyzer turns from a testing tool into a development tool such as the text editor, compiler, etc.

The combined use of the proposed methods can be the further development of the optimization of the project analysis time. For example, the trace elements can be played lazily or the unused objects in headers may be skipped while the incremental analysis is performed. To eliminate the repeated analysis of the same source code, it is important to reduce the size of precompiled headers. Then, they can be created automatically while analyzing the files and the loading time of the precompiled headers is reduced.

REFERENCES

1. Avetisyan, A., Belevantsev, A., Borodin, A., and Nesov, V., The Use of Static Analysis for Detecting Vulnerabilities and Critical Bugs in Source Codes, in *Trudy ISP RAN,* 2011, vol. 21, pp. 23–38.
2. Syromyatnukov, S.V., Declarative Interface for Bug Detection Based on Syntax Trees: The KAST Language, *Trudy ISP RAN*, 2011, vol. 20, pp. 51–68.
3. www.klocwork.com
4. MS Visual Studio Precompiled Headers, http://msdn.microsoft.com/en-us/library/szfdksca(v=vs.71).aspx
5. GCC Using precompiled headers, http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html
6. MSDN Template Specifications http://msdn.microsoft.com/en-us/library/x5w1yety%28v=vs. 71%29.aspx
7. Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques, and Tools*, Reading, Mass.: Addison-Wesley, 1986.
8. http://gcc.gnu.org/wiki/pph