BACKWARD COMPATIBILITY VIOLATIONS AND THEIR DETECTION IN C++ LEGACY CODE USING STATIC ANALYSIS

Tibor BRUNNER, Norbert PATAKI, Zoltán PORKOLÁB
Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary,
E-mail: {bruntib, patakino, gsd}@caesar.elte.hu

ABSTRACT

Programming languages evolve as the need for higher abstraction level increases. To satisfy these needs, languages introduce new features, which are usually additional elements, but many times these are not orthogonal to the existing ones. The C++ programming language is planned with backward compatibility kept in mind between particular releases. However, some of the new features introduced by C++11, like move semantics or multi-threading required the change of the standard library API and this leads to deviation in the meaning of existing programs. In this article we draw the attention on some of these semantic changes and provide a tool for automatically detecting them.

Keywords: *Abstract syntax tree, C++, static analysis*

1. INTRODUCTION

Programming languages are human made artefacts; they are created to solve certain problems in a certain time in a certain environment. They are rarely perfect in their first version, it is typical that they change frequently and significantly in their early period as experiences are accumulated and developers face new problems. Although such changes are inconvenient, they are usually manageable as the existing code base written in a relatively new programming language is typically not huge.

Mature languages change less frequently. For mainstream languages such changes should be planned in a very careful way to avoid the invalidation of the existing large code base. Therefore, the modifications are typically *backward compatible*, i.e. the earlier written code should compile with the new version of the compiler and the *meaning* of the old code should not change. This is a major requirement as it is unrealistic that the maintainers of a large code base execute a full code review on the source to detect whether certain code parts are affected by the changes [1].

If the evolution of the language requires backward incompatible changes, the usual procedure is the following: first, the affected language element becomes *deprecated*, i.e. it exists in the current language version but not suggested to use it. Deprecated language elements are easily recognizable as the compiler emits diagnostic messages over them. Maintainers should change the source code as soon as possible to eliminate the usage of deprecated elements. Later, the element can be officially removed from the language and referring it should cause syntax error.

The most serious problems related to language evolution come from situations where the syntax of the old language version remains valid in the new version but the *semantics* i.e. the meaning of the source code changes. In such cases no compiler diagnostics are produced, nothing warns the maintainers for the dangerous difference. The source code is recompiled with the new compiler version supporting the new language version without error messages or even warnings. When the program will be executed, however, one can recognize a completely different result. To catch such situations is very dubious even when an almost full regression test coverage exists. Naturally, mainstream languages try to avoid such *backward incompatible semantic changes*. However, even with the best intentions, such situations happen regularly.

The C++ programming language [2, 3] is a favourite choice of implementation when performance (maximizing the speed of the program, minimizing memory consumption or power supply) is among the major decision criteria. Large systems in telecommunications, aeronautics or other high speed programming areas are frequently written in C++. In such systems object oriented and generic programming features are mixed in a highly complex way.

The C++ programming language has a long history. The language originated in C and one of the early design goals of C++ was to provide some sort of backward compatibility with C [1]. Although this concept became less important later, it helped C++ to reach the critical amount of developers with the reuse of old C libraries. C++ as a *safer C* added mostly new features to C which did not interfere with existing language structures. The standardization process of the language in the second part of the 90s led to the first international C++ standard, frequently referred as C++98.

In the next few years some of the shortages and bugs of that standard has been revealed and a new version of the standard (often mentioned as C++03) issued mainly as a correction of the original standard. This version contained only minor changes and had no drastic effect on existing C++ systems.

Since the standardization, the popularity of C++ has been continuously increasing. Millions of code lines have been written in large software systems in various application areas. That means, we have now a large amount of legacy code with all the maintenance issues connected with them. Regarding this large code base any change in the core language or in the standard library requires a special attention.

A few important features, however, was left out from C++03 [4]. Library elements like hash tables [5] and smart pointers [6] among others were supposed to be selected part of the planned next standard referred as C++0X. However, some of the proposed changes was affected the core lan-

guage, like *move semantics* [7, 8]. As the new standard, called C++11 has been implemented in more and more compilers, people attracted by the new features started to use these features, believing that the semantics of old existing code will not change.

Unfortunately, recently we identified a number of serious defects in backward compatibility between current and previous C++ standards. This means, that the recompilation of the previously written code using a new C++ compiler (using the new standard) *may change the program behaviour* without any compiler diagnostic messages. Needless to say, that such changes in program semantics can be extremely dangerous and in a large code base they are almost impossible to catch by manual inspection (like code review). Whether regression tests can capture the problems is also questionable.

In this paper we propose a tool support to detect such backward incompatibilities in C++. The basic idea is to parse the source code using different standards, and then compare the generated abstract syntax trees (AST), and reporting significant differences.

We implemented a proof of concept prototype tool to demonstrate our idea. The tool is able to compare C++98/03/11/14 versions of a program and detects parsing differences. The prototype tool is implemented using the LLVM/Clang compiler infrastructure and is available for tests.

This paper is organized as follows. We demonstrate the most important changes that may cause backward incompatibilities between programs compiled in different C++ standard versions and give examples for such situations in Section 2. We describe the Clang compiler infrastructure as the basis of our solution in Section 3. Our prototype tool is introduced in Section 4. In Section 5 we overview our results produced by our tool. A possible alternative solution is discussed in Section 6 and compared to our method. The paper concludes in Section 7.

2. MAJOR CHANGES IN C++11/14

A software quality has several aspects such as performance, low memory usage, portability, safety, robustness, and so on. Some of these aspects may be contradictory, for example a program can be faster if the computed data is cached, but in this case more memory is needed. If the values are computed over and over without storing the results, the program needs less memory but the performance decreases. The programming languages have to choose which aspects to support, and which not. In many cases, where in doubt, ADA chooses safety, Java chooses portability and C++ chooses runtime performance. Along the development of a programming languages their primary principles are always kept in mind, and the new features also intend to serve them.

Object-oriented languages are designed to give programmers the possibility of creating new types in the form of classes. Such types may be constructed from other complex types in a recursive manner and copying their raw bytes may not be the proper copy semantics. For such cases programmers in C++ may define *copy constructor* for initialization and the `operator=` for assignment. These *special member functions* are usually implemented using the already defined copy operations of subobjects or the explicit copy instructions decided by the programmer. When no user defined copy constructor or assignment operator have been provided the default memberwise copy operations will be applied.

C++ was designed using *value semantic*, i.e. every object owns its memory location uniquely [9]. Although, this rule can be violated using pointers and the address-of operator, by default C++ assignment copies the object to a new location. For user defined types the programmer can define his own copy constructor and assignment operator. While this behaviour is very convenient when we want to encapsulate the implementation of classes and building higher abstractions, it may also be a cause of serious performance issues. In case of complex classes, like matrices, vectors, lists, a single assignment may cascade down to a huge number of byte-level copies. This phenomenon exists for the containers of the C++ standard library too. Temporary objects created during the evaluation of expressions are also critical performance bottlenecks. The used `new` and `delete` operators, the overhead of the extra loops and memory access are costly operations.

Todd Veldhuizen investigated this issue, and suggested C++ expression templates [10] and template metaprogramming [11, 12] as a solution. The basic idea is to avoid the creation of temporaries, and unnecessary copies, but "steal" the resources of the operands, i.e. move the ownership of the data representation between assigned objects. Such *move* operations can be implemented library-based, like the `Boost.Move` library [13] with overloading the original *copy* operations. Library-based solutions, however, lack to distinguish objects which are destroyable, i.e. their resources can be safely moved out. To distinguish non-destroyable objects from possible sources of move operations, i.e. those which can be destroyed language support is required.

The C++11 standard has introduced a new reference type: the *rvalue* reference [2, 3] and change the core language to enable *move semantics*. In the source code, the rvalue references have a new syntax `&&` to yield reference to destroyable objects. Using this syntax, constructors, copy constructors and assignment operators can be overloaded with multiple types. Constructors and assignment operators with rvalue parameters are called *move constructor* and *move assignment*. The move constructor and move assignment changes the ownership of the data representation of the argument object, and set it to an empty but valid (destroyable) state [14].

To understand move semantics let us take a look at the following example:

```
Vector a, b, c, d;
d = a + b + c;
```

When `a + b` evaluates, a temporary object (`t`) is created to store the result. In the next step `t + c` is evaluated which produces another object. The disadvantage is obvious. For computing the result of each sum, a newly created object is needed for storing the result. These are tempo-

rary objects which are deallocated as soon as the summation ends. Allocation and deallocation of large objects is expensive. It would be much more efficient if these temporary objects could be reused between the operations, and this can be expected from a language of which the superior philosophy is the support of performance.

The problem comes from that before C++11 standard it was not possible to distinguish between temporary objects and long-term objects. In C++11 not only the type of an expression is important but the category as well. Besides typical *lvalue* and *rvalue* categories the *xvalue* has appeared that indicates the values of which the resources can be reused, or to use standard terminology, can be moved from. For sake of simplicity let us suppose that a `Vector` is represented by an array of elements and by their number. Moving from a `Vector` object to another means that only the pointer indicating the first element's location of the source `Vector`'s array is copied to the target `Vector`'s inner representation, not every elements of the array. The rule is that the source object from which we move has to remain destructible, since the destruction of an object cannot be prevented even if it is a temporary object.

Returning to the example of summation, the implementation of the `operator+` could be improved so if it has a temporary object as first parameter then instead of creating a new result object the resources of the temporary object would be reused. The C++11 standard introduces the concept of *right value reference* which matches the temporary objects (i.e. the objects which cannot be reached by name or made explicitly right value reference by `std::move()` function. A right value reference for type `T` is written `T&&`.

```
Vector operator+(Vector&& l, const Vector& r);
```

3. TECHNICAL BACKGROUND

In this article we are dealing with the semantic changes of our C++ programs between the different standard versions. Since many semantic differences are mostly raised by introducing new language elements or by modification of the Standard Template Library (STL) which is an integral part of C++ language, these changes come with the change of Abstract Syntax Tree (AST). Here we are considering a simple tree data structure which is equipped with information carrying the meaning of our programs, so the differences can be checked algorithmically.

C++ is a relatively complex language. The complicated syntax should denote differences between names declared in different namespaces or other contexts. Overloading resolution – especially when speaking about *argument dependent lookup* (ADL) – and scope resolution use highly elaborated algorithms. Template instantiations, template specialisations and SFINAE rules are also unobvious. There is no surprise that only a few C++ compilers conform to the recent standards.

Clang is a C++ compiler which is considered as the most compatible one to the standard. It was intentionally designed as a set of reusable libraries with well defined API to allow programmers to handle the different parts of compilation. In each step of compilation (lexical analysis, preprocessing, compilation, optimization, code generation) the various representations of the source code arise [15]. In this article we conclude our results about the semantic changes of a program based on the *abstract syntax tree* (AST), thus in this section we take a closer look at Clang abstract syntax tree representation.

The *abstract* word suggests that the tree does not keep every single character of the source, but only the meaningful elements of the programming language. For example the node which belongs to a `for` loop has four children: a declaration statement to introduce the loop variable, a logical expression as loop condition, an iteration expression and the body. Note that the parentheses and the semicolons in the loop header are excluded.

In the AST there are different type of nodes such as `ForStmt`, `FunctionDecl`, `BinaryOperator`, etc. These types are organised to an inheritance hierarchy which has three roots: `Decl`, `Stmt` and `Type`. Since the fundamental part of build process is compilation of translation units, the type of the root node is `TranslationUnitDecl`.

One way of using the Clang AST is to visit its nodes [16]. The visitor design pattern can be used to reach every node of the tree and perform some action when the process comes to a given type of node. Clang compiler provides a very efficient way of tree traversal by `RecursiveASTVisitor` template class. Our visitor class has to inherit from this template class of which the template parameter is our class itself. The reason of this is that with this solution our class also becomes an AST visitor by the inheritance, but we do not have to pay for virtual function calls every time when running the given visitor function for the next AST node.

Each node type has a corresponding visitor function which is called exactly the same as the node type prefixed by "`Visit`". This means that when the depth-first traversal provided by `RecursiveASTVisitor` reaches a function declaration then `VisitFunctionDecl()` function is run. As mentioned above, this isn't a virtual function, so appending `override` keyword at the end of this function generates a compiler error. One may notice that by visiting a function declaration not only `VisitFunctionDecl` runs but `VisitDeclaratorDecl`, `VisitValueDecl`, `VisitNamedDecl` and `VisitDecl` too. This happens because besides these visitor functions the various node types also have a "`Traverse...`" method of which the default behaviour is to call the "`WalkUpFrom...`" function, that calls the corresponding visitors along the inheritance hierarchy. Just like "`Visit...`" and "`WalkUpFrom...`", "`Traverse...`" can also be rewritten but in this case the parent's traverse method has to be called explicitly to do the walk up. Detailed documentation of `RecursiveASTVisitor` (in Doxygen format) can be found in [17].

A visitor function is passed a pointer to the visited AST node. This allows us to perform arbitrary node-based operation. Via this pointer it is possible to reach any information about the node which could be queried during the compilation phase, like the position of the language element in the source code, the callee of a function call, the *then* and *else* branches of an `IfStmt`, etc.

```
‘-FunctionDecl 0x2eff420 <line:7:1, line:10:1> line:7:5 main ’int (void)’
  ‘-CompoundStmt 0x2f12380 <line:8:1, line:10:1>
    ‘-DeclStmt 0x2f12368 <line:9:3, col:23>
      ‘-VarDecl 0x2eff948 <col:3, col:22> col:18 v ’std::vector<C>’:’class std::vector<class C, class std::allocator<class C> >’ callinit
        ‘-ExprWithCleanups 0x2f12350 <col:18, col:22> ’std::vector<C>’:’class std::vector<class C, class std::allocator<class C> >’
          ‘-CXXConstructExpr 0x2f12308 <col:18, col:22> ’std::vector<C>’:’class std::vector<class C, class std::allocator<class C> >’
                ’void (size_type, const value_type &, const allocator_type &)’
            |-ImplicitCastExpr 0x2f11ae8 <col:20> ’size_type’:’unsigned long’ <IntegralCast>
            | ‘-IntegerLiteral 0x2eff9a8 <col:20> ’int’ 10
            |-CXXDefaultArgExpr 0x2f11f08 <<invalid sloc>> ’const value_type’:’const class C’ lvalue
            ‘-CXXDefaultArgExpr 0x2f122e0 <<invalid sloc>> ’const allocator_type’:’const class std::allocator<class C>’ lvalue

‘-FunctionDecl 0x2f12890 <line:7:1, line:10:1> line:7:5 main ’int (void)’
  ‘-CompoundStmt 0x2f4f0b8 <line:8:1, line:10:1>
    ‘-DeclStmt 0x2f4f0a0 <line:9:3, col:23>
      ‘-VarDecl 0x2f12da8 <col:3, col:22> col:18 v ’std::vector<C>’:’class std::vector<class C, class std::allocator<class C> >’ callinit
        ‘-ExprWithCleanups 0x2f4f088 <col:18, col:22> ’std::vector<C>’:’class std::vector<class C, class std::allocator<class C> >’
          ‘-CXXConstructExpr 0x2f4f048 <col:18, col:22> ’std::vector<C>’:’class std::vector<class C, class std::allocator<class C> >’
                ’void (size_type, const allocator_type &)’
            |-ImplicitCastExpr 0x2f4ef68 <col:20> ’size_type’:’unsigned long’ <IntegralCast>
            | ‘-IntegerLiteral 0x2f12e08 <col:20> ’int’ 10
            ‘-CXXDefaultArgExpr 0x2f4f020 <<invalid sloc>> ’const allocator_type’:’const class std::allocator<class C>’ lvalue
```

Fig. 1 AST dumps of `main()` functions generated by Clang

4. OUR CONTRIBUTION

Static analysis tools gain more and more popularity. The reason is that they can find a high number of possible dangerous program constructs with a relatively low effort. Compared to the various testing approaches they do not require a high runtime coverage or manual work on development test cases.

Static analysis methods vary on complexity and precision level [18]. Some of them convert the source code to normal form and perform a simple regular expression matching on it to find given patterns. This is suitable only for the simplest cases. Other tools perform pattern matching directly on the AST. This method can take semantic details into account. There are even more sophisticated techniques that rely on program dependency graphs [19] or with symbolic execution which interprets the code thus providing dynamic information on the program behaviour [20]. The question arises as to whether it would be enough to use such tools to look for the specific cases where semantics change from a standard version to another. The disadvantage of this approach is that these tools can notice only those type of changes which they are prepared for. Our solution should discover every difference as it examines every queriable information about the syntax trees.

When we decided to create a prototype program to detect semantic differences between different programming language versions we had to choose between the options above. We have to decide a trade-off between the completeness of the analysis and the performance restrictions, i.e. our tool should be applicable for large legacy code base of million lines of C++ code. Based on these constraints we have chosen the AST-based solution.

Our prototype is written in standard C++ applying features of C++11. The program uses the Clang compiler as a library for building the AST. This is done by filling all the visitor functions to dump the nodes of the AST to a database. For this purpose we use SQLite database for sake of simplicity.

The tree structure itself can be stored in a traditional way in the database as parent-child pairs. During the tree traversal we can extract this information by the `TraverseDecl()` and `TraverseStmt()` functions of `RecursiveASTVisitor` class. In these methods the currently traversed node is placed on the top of a stack data structure to keep the invariant that the visited nodes after the placement are the descendants of this node. After the traversal of the subtree under the top element, the node is removed.

In the Clang compiler all nodes are identified by the object of the node itself, so a pointer to the node uniquely determines the given entity. This cannot be used as ID in our case, since every run of the program may place the nodes to different locations in the memory, not to mention that compiling the same source code with different standards may result completely different trees. Fortunately identifying the nodes is not necessary, since we are only taking into account the tree structure and the descriptive information belonging to the nodes. We say that two nodes are uniform if every information that we can observe about it completely match. Thus it is not important to know whether the corresponding nodes are identical, it is enough if their properties are the same. For this reason the database contains a table in which the properties of a node are stored. Besides the usual ID column, this table has three other fields for the node ID, the property name and the property value. During the visiting process when reaching a node we store every information that can be queried about it by getter functions. The getter function name is stored as the property name and the returned value is stored as the property value. These values are like the mangled name or the parameter list of a function declaration, or the loop condition and iteration expression of a `for` loop.

By running the program for example in C++98 and C++11 mode, the whole syntax trees are persisted in a database. The last thing to do is to compare them. For comparison we use a simple depth-first search. We iterate the two trees in parallel from the root belonging to the Translation Unit kind node, and in each step we select the properties of the current node. If any difference is found then iteration stops and the incidence is being reported. Figure 1 shows the ASTs of the `main()` function of the example demonstrating the semantic differences of `vector` construction with initial elements, using C++03 and C++11 respectively.

5. RESULTS

In this section we summarize the results we experienced while testing a number of C++ sources. We found various possible regression patterns, some of them may cause serious changes in the program semantics.

The first thing that our tool detected was about standard library supplements. The standard library is an integral part of modern programming languages, which makes its usage easier. The release of a new version affects the standard library as well: new elements appear or vanish, and the existing ones change. In C++ these changes are controlled by `__cplusplus` macro. This is a macro token which contains version information about the current compiler and when used in a library, this defines in the preprocessing phase which parts of the library file should be included in the final program code. This way it is possible in C++ to conditionally compile specific parts of the program. For example if the compilation is done in C++11 then `__cplusplus` is automatically set to 201103L. Source code written between `#if __cplusplus >= 201103L` and `#endif` are parts of the program only if this condition evaluates true. Among others this makes function overloads on newly introduced `long long int` type visible. Of course this does not lead to any program fails, because until C++11 this type was not available.

Some more advantageous changes happen based on move semantics. If a class is created for demonstrative reasons with default and copy constructor, assignment operator and destructor with a message print to the standard output, then we can inspect what happens when a temporary object of this type is inserted to an empty *map* by `operator[]`. Until C++11 this operation invoked default and copy constructors twice, copy assignment once and the destructor four times. Their order differs in GCC and Clang, but the number of invocations is the same. Compiling the same program in C++11 mode the output shows two default constructor and a copy assignment invocation. Note that no compiler generated move operations occur in this case, because the standard states that move constructor and move assignment operator is not generated when user defined copy operations are present. However we can now implement move constructor and assignment in which case copy assignment is replaced by move assignment [14].

These changes are not necessarily risky. In those cases where programmers don't rely on side effects, global and static variables, then using less constructors and using move operations instead of copy just makes the code faster. Strings, vectors, sets, maps, etc. work automatically much faster when the program is compiled in C++11 mode or above. However, we can present examples where turning on C++11 compatibility the program shows off observable differences as the semantics changes.

There are some types which do not support copy operations, such as `std::ifstream` or `std::thread`. However, moving them is possible. Say we want to create a thread pool, a simple vector of threads. `std::vector<T>` has a constructor which enables allocation of *n* objects of type `T`. Until C++11 this constructor has expected the number of elements *n*, and an object of type `const T&` with a default constructed default value, and an allocator (this latter one is not important for now). This constructor copies *n* instances of the second parameter. Move semantics has motivated the C++ standard committee to reconsider this constructor as it is not suitable for creating *n* instances of those types which do not provide copy operations. Thus the signature of the constructors has changed: there is a new constructor function with exactly one parameter: the number of elements. This function instantiates *n* default constructed objects and places them into the vector. If one needs objects other than default constructed then he or she has to use the constructor with two parameters where the second one does not have any default value. This copies the passed object to the *n* places.

Consider the following code snippet:

```
struct S {
  S() : i(++counter) {}
  static int counter;
  int i;
};

int S::counter = 0;

int main() {
  std::vector<S> v(5);
  for (std::size_t x=0; x<v.size(); ++x)
    std::cout << v[x].i;
}
```

The program above has different semantics in C++11 and pre-C++11 versions. The output of the former one is `12345` while the latter one prints `11111`. Before C++11 a single object of type `S` has been created by the default parameter of the constructor, and this instance has been copied *n* times into the vector. From C++11 the constructor with one parameter has been invoked and it is implemented to call the default constructor of type *S* for *n* element of the vector.

One might criticise the designer of the class *S* as the copy and the move operation behave differently. However, there are situations where this is the required behaviour. Also, in practical code, side effects can happen in copy and move operations (like related actions for persistence). Even if this design would be considered questionable, this kind of construct occurs in real world applications. As the change happened outside of the user written code (in the implementation of the standard library), it will be very hard to detect by manual inspection. Therefore this kind of semantic changes are considered extremely harmful.

There are also some curiosities around the preprocessor. One can define macro tokens which can be considered roughly as textual replacement operations. Each occurrence of these in the source code is replaced at the very beginning of the compilation process. Macros form a sub-language which does not handle types or values. The syntax of this is `#define <from> <to>`. For example we could define `u8` to `"abc"`:

```
#define u8 "abc"
std::cout << u8"xyz";
```

In the second line the `"u8"` is substituted by `"abc"` and this forms a string concatenation. The result is that `abcxyz` is printed to the output. In C++11 `"u8"` has a special mean-

ing. Writing this in front of a string literal that will considered to be an UTF-8 encoded character sequence.

Introduction of user defined literals can lead to similar problems. In case of `"hello"_x` the `_x` can be defined as a macro as above or there might be an `operator""_x()` which has different meaning.

We can also provide an example which differs in its performance:

```
A f(int i) {
  A x, y;
  if (i > 1) return x;
  else       return y;
}

int main(int argc, char *argv[]) {
  A z = f(argc);
}
```

The standard states that if a function returns an automatic variable of which the type equals the return type of the function, then the compiler may apply a so called *return value optimization* (RVO) [21]. In that case the returning object can be created on the stack at the place of the function invocation, and the copy construction can be omitted even if it has any side effects. However in some cases like the example above the compiler cannot predict which local object has to be created at the place of function call and in this case the compiler cannot avoid the creation and copy of the given local variable. From C++11 instead of calling copy constructor, the compiler considers the return statement as if an implicit `std::move()` were placed around the returned object thus moving it to its target.

The above-mentioned behavioural changes affect the modification of the syntax tree. In case of introduced new types like `long long int` the AST is obviously extended with the function overloads of which the signature contains this type. The appearance of move semantics introduces a new value category, the right value references. Function overload on this is allowed in the language. In the UTF-8 example two different string literals are generated, and these are also observable in the AST. Our tool is able detect all these differences, so it is possible to report semantic changes to the users.

We note that in some cases the reports may be false positive. For example in the general case it is not appropriate to rely on side effects in the implementation of a constructor. In the example where the potential semantic change was caused by the changing of `std::vector`'s constructor our tool gives a false report if there is no semantic difference in the two cases.

Theoretically, a possibility for false positives comes from changed internal data structures or algoithms. These cases, however, can be considered as false positive cases only when the *observable behavior* is exactly the same, only the implementation has changed. In practice, we have not experienced such cases. The reason is that the complexity of the C++ programs – especially in multithreaded environment – discourages developers to execute such modifications.

The compilation of C++ programs contain the step of building the abstract syntax tree – the fundamental task of our utility. However, our tool does not include time consuming steps like dataflow analysis, code optimization, code generation and linking. On the other hand, our tool has a full walk-through of the AST. Overall, these factors are in ballance. Our tool runs roughly as long as a normal compilation process.

6. RELATED WORK

An alternative solution for the problem discussed in this paper would be to use static analysis *design rule checkers*. Design rule violation detection is an emerging technique to improve the quality of large scale software and reduce the implementation – testing – bug fixing cycle.

Tools for design rule violation detection for C++ code exist in a growing number but their power are different. The most simple checkers using simple *pattern matching* using regular expressions on the canonical format of the source code. Such tools, like [22] are popular since they are relatively fast and new rules can be implemented in an easy way just defining the required new regular expression [23]. However, as regular expressions are context-free languages, the theoretical detection power of these tools are limited.

As an example, suppose, that we want detect the improper usage of the `sizeof` operator, e.g. when we applied `sizeof(ptr)` for a pointer instead of the object itself, like `sizeof(*ptr)`. This is likely a bug as pointers in a certain platform have fixed size independently of the pointed object. We might detect specific patterns in the code, like `sizeof(&x)`, but in the generic case of `sizeof(x)` we can not investigate the *type* of the `x` the argument, since that depends on the declaration, which is placed somewhere else in the code.

More complex tools are working on the abstract syntax tree [24]. These tools can walk on the AST as our solution does it, can refer to the point of definition of the program elements, and can recognize their types. The major problem with this approach is that they require to implement all the possible design rules to check. This is not only a giant work, there is also a theoretical problem: how can we detect the backward incompatibilities caused by issues that we are not aware yet?

We think our approach as superior over the design rule violation detection since we do not have to specify in advance which kind of problems we are looking for. We simply compare the two versions of the program and anywhere we find a difference we report.

7. CONCLUSION

In this paper we warn for the danger of backward incompatible changes in mainstream programming languages and suggested a tool-based solution for C++ legacy code base. In recent C++ standard, C++11 and C++14 there are some changes both in the core language and in the standard library when such situations can occur.

We discussed the details of programming language evolution from the viewpoint of the backward compatibility between different language versions. We identified a specially harmful situation when the earlier written code is recompiled in the new language standard with the new compiler

version without compiler diagnostics, but its semantic, i.e. the program behaviour changes. Such situations are very hard to detect manually, especially in the case of recompilation of a large legacy code base.

The C++ programming language is among the most popular programming languages where large legacy code base has been developed for decades. Recent major evolutionary steps caused extensive changes both in the core language and in the standard library. Unfortunately some of them break the backward compatibility. We explored such cases related to the move semantics and other language features. To assist the automatic detection of these kind of problems we suggested tool support.

We developed a proof of concept to validate our idea. The tool is based on the open source LLVM/Clang infrastructure, which is a reusable library supporting – among others – to parse the source code and travel the resulted abstract syntax tree. We parse the same source using different compiler version settings (according to different language versions) and compare the ASTs. Any semantic differences should materilaize in a difference between these trees. We report these differences using a pretty printing function referring the differences in a usef-friendly form.

For all static analysis tools there is the possibility for false positive findings. Our tool is not an exception. However, in practice we experienced low level of false positives – a result likely reflects the defensive approach of the C++ language designers. The tool has no run-time overhead compared to the normal compilation process.

We believe that our tool gives a major help for those developers, who are maintaining large legacy C++ code base. As such code bases are continuously developing, the new features are implemented and compiled using new compiler versions if and only if we can ensure that the earlier written code – if will be compiled – will keep its semantic, i.e. it will behave like compiled in the earlier compiler versions. Our tool can help to prove this.

ACKNOWLEDGEMENT

The authors would like to thank the Doctoral School of the Faculty of Informatics of Eötvös Loránd University and Ericsson Hungary Ltd. to support this research.

REFERENCES

[1] STROUSTRUP, B.: *The Design and Evolution of C++* Addison-Wesley Professional, ISBN-10: 0201543303, ISBN-13 078-5342543308, 1994.

[2] STROUSTRUP, B.: *The C++ Programming Language, 4th ed.*, Addison-Wesley, ISBN 978-0321563842, 2013.

[3] *ISO International Standard*, ISO/IEC 14882:2011(E) – Programming Language C++, 2011.

[4] STROUSTRUP, B.: *The design of C++0x*, C/C++ Users Journal, May 2005.

[5] AUSTERN, M.: *Proposal to Add Hash Tables to the Standard Library (revision 4)*, N1456, Programming Language C++, Library Subgroup, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html, 2005.

[6] DIMOV, P. – DAWES, B. – COLVIN, G.: *A Proposal to Add General Purpose Smart Pointers to the Library Technical eport*, N1450, Programming Language C++, Library Subgroup, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html, 2003.

[7] BECKER, T.: *C++ Rvalue References Explained*, http://thbecker.net/articles/rvalue_references/section_01.html

[8] HINNANT, H. E. – DIMOV, P. – ABRAHAMS, D.: *A Proposal to Add Move Semantics Support to the C++ Language*, N1377, Programming Language C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm, 2002.

[9] ELLIS, A. M. – STROUSTRUP, B.: *The Annotated C++ Reference Manual* Addison-Wesley Professional, ISBN 0-201-51459-1, 1990.

[10] VELDHUIZEN, T.: *Expression templates*, C++ Report, 7(5):26–31, 1995.

[11] ALEXANDRESCU, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, Boston, MA, ISBN-13: 078-5342704310, ISBN-10: 0201704315, 2001.

[12] VELDHUIZEN, T.: *Using C++ template metaprograms*, C++ Report, 7(4):36–43, 1995.

[13] ABRAHAMS, D. – GURTOVOY, A.: *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, MA, ISBN-13: 978-0321227256, ISBN-10: 0321227255, 2004.

[14] MEYERS, S.: *Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st edition, O'Reilly, ISBN-13: 978-1491903995, ISBN-10: 1491903996, 2014.

[15] HORVÁTH, G. – PATAKI, N.: *Clang matchers for verified usage of the C++ Standard Template Library*, Annales Mathematicae et Informaticae 44, pp. 99–109.

[16] MÁJER, V. – PATAKI, N.: *Concurrent object construction in modern object-oriented programming languages*, Proc. of the 9th International Conference on Applied Informatics (ICAI), Vol. 2, pp. 293–300.

[17] *Doxygen documentation of recursive AST visitors*, 2016, http://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html

[18] WAGNER, S. – JÜRJENS, J. – KOLLER, C. – TRISCHBERGER, P.: *Comparing bug finding tools with reviews and tests*, TestCom'05 Proceedings of the 17th IFIP TC6/WG 6.1 international conference on Testing of Communicating Systems, pp. 40-55, Springer-Verlag Berlin, Heidelberg ISBN:3-540-26054-4 978-3-540-26054-7, 2005.

[19] HARROLD, M. J. – MALLOY, B. – ROTHERMEL, G.: *Efficient construction of program dependence graphs*, In ISSTA '93 Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, pp. 160-170, ACM New York, NY, USA 1993 ISBN:0-89791-608-5

[20] HALLEM, S. – CHELF, B. – XIE, Y. – ENGLER, D.: *A System and Language for Building System-Specific, Static Analyses*, SIGPLAN Not. 37, 5 (May 2002), 69-82. DOI=http://dx.doi.org/10.1145/543552.512539

[21] MEYERS, S.: *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd edition, O'Reilly, ISBN-13: 078-5342334876, ISBN-10: 0321334876, 2005.

[22] MARJAMAKI, D.: *CppCheck – A tool for static C/C++ code analysis*, http://cppcheck.sourceforge.net/, 2016.

[23] MOENE, M.: *Search with CppCheck*, Overload Journal, Vol. 120. http://accu.org/index.php/journals/1898, April 2014.

[24] KRUPP, D. et. al.: *CodeChecker – a static analysis infrastructure built on Clang Static Analyzer*, https://github.com/Ericsson/codechecker, 2016.

[25] CZARNECKI, K. – EISENECKER, U.: *Generative Programming: Methods, Tools, and Applications*, ACM Press/Addison-Wesley Publishing Co., ISBN 0-201-30977-7, 2000.

[26] KOLPACKOV, B.: *The Clang AST – a Tutorial*, http://www.codesynthesis.com/boris/blog/2012/04/18/cxx11-generalized-attributes/, 2012.

Received April 18, 2016, accepted July 29, 2016

BIOGRAPHIES

Tibor Brunner was born on 2. 7. 1988. In 2013 he graduated (MSc) at the Faculty of Informatics at Eötvös Loránd University in Budapest. He started his PhD study in the field of programming languages and paradigms in 2015. His main research area is programming languages evolution and the modern features of the multiparadigm languages. He participate in the teaching of programmer students at B.Sc. level.

Norbert Pataki was born on 26. 2. 1982. He received his PhD in 2013 at the Eötvös Loránd University, Budapest. The title of his thesis is "Correctness of Generative Programs". He teaches C++ programming language courses at the university since 2005. He participates in teaching of project tools as well. His main research areas are C++, programming languages, software metrics, generic and generative programming, static analysis and DevOps.

Zoltán Porkoláb was born on 11. 11. 1963. He defended his Ph.D. in 2003 at the Eötvös Loránd University, Budapest, where he is an associate lecturer in the Department of Programming Languages and Compilers. Meanwhile, he is Principal C++ Developer in Ericsson Hungary Ltd. His research focus is programming languages, especially the C++ programming language. He was the leader of the Hungarian translation of Bjarne Stroustrup's book The C++ Language, Special Edition.