Transforming C++11 Code to C++03 to Support Legacy Compilation Environments

Gábor Antal, Dávid Havas, István Siket, Árpád Beszédes, Rudolf Ferenc
Department of Software Engineering
University of Szeged, Szeged, Hungary
{antal,havasd,siket,beszedes,ferenc}@inf.u-szeged.hu

József Mihalicza
NNG LLC
jmihalicza@gmail.com

***Abstract*—Newer technologies – programming languages, environments, libraries – change very rapidly. However, various internal and external constraints often prevent projects from quickly adopting to these changes. Customers may require specific platform compatibility from a software vendor, for example. In this work, we deal with such an issue in the context of the C++ programming language. Our industrial partner is required to use SDKs that support only older C++ language editions. They, however, would like to allow their developers to use the newest language constructs in their code. To address this problem, we created a source code transformation framework to automatically backport source code written according to the C++11 standard to its functionally equivalent C++03 variant. With our framework developers are free to exploit the latest language features, while production code is still built by using a restricted set of available language constructs. This paper reports on the technical details of the transformation engine, and our experiences in applying it on two large industrial code bases and four open-source systems. Our solution is freely available and open-source.**

***Index Terms*—C++, source code transformation, legacy systems, language backporting**

I. Introduction

Today, technologies used in software engineering practice, such as programming languages, environments and libraries, change on an unexperienced pace. And, naturally, developers would like to exploit the advantages of such developments in order to increase their productivity, quality of code and reduce risks of error. However, often there are certain constraints in the projects that prohibit using the newest technologies. This includes, for instance, interoperability with legacy systems, compatibility with older hardware and software, and other limitations arising from the context of the project. For instance, in a situation when the software vendor delivers software to a customer, it must conform to the customer's requirements regarding platform compatibility.

The work presented in this paper was motivated exactly by such a situation. NNG LLC, our industrial partner, is a company that develops navigation software, and as such it delivers software products to its clients who integrate the navigation software component into the host system of the final product. These host systems often raise strict technical constraints against the delivered software to be integrated. Compatibility may be required with old operating systems, libraries, and existing components. Consequently, the development company needs to enforce strict regulations in-house regarding the usable platforms, language versions and development environments. The net effect is that the developers are confronted with a situation in which they are limited by older technologies, while they would be eager to use more advanced ones. Often, this leads to lower productivity and even lack of motivation because their professional skills development is limited as well.

In this work, we deal with the mentioned problems in the context of the C++ language, the primary technology used by the company. For many years, the official language standard has not been updated until 2011, which progressively resulted in the birth of a large code base globally, which is now treated already as legacy code. The C++11 standard [13] included so many new features (such as in-class initializations, lambda functions, automatic types, attributes, and many more) that made it almost a new language (even Bjarne Stroustrup, the creator of C++ thinks it "feels like a new language[1]"). However, even after five years of the publication of the new standard, developers at NNG are still forced to use older versions of the language, which is a significant drawback from both the subject system and from the developers' point of view.

Hence, the goal of our R&D collaboration project was to develop a solution to this problem in a way that would be both beneficial for the developers and the system itself. We created a source code transformation framework with which C++ source code written according to the C++11 standard can be automatically "backported" to C++ code conforming to earlier language versions (C++03, in particular [12]). The framework is capable of automatically transforming a large number of new language constructs to their equivalent versions in the older language. This way, developers are free to exploit the latest language features, while production code is still built by using a restricted set of available language constructs. Even though various technical limitations prevented us from making a complete transformation solution in terms of supported language elements, our framework enables a very large subset of C++11, making it usable in practice.

The transformation framework includes a number of additional features besides transforming individual source code files, which make its integration into practical build processes easier. These include, among others, source tree mirror-

[1] http://www.stroustrup.com/C++11FAQ.html\#think

ing, incremental transformation, selective transformation, and traceability between the original and the transformed code. The technology has been experimentally integrated into the development process of the company (which was not trivial due to some unique properties of the build process), enabling them to benefit from using recent technology while retaining compatibility with their partners using legacy systems.

This paper reports on the technical details of the transformation engine, and our experiences in applying it not only on NNG's code base but on another industrial application and on four open source systems as well. Although the transformations do not cover C++11 in 100%, our results and experiences with industrial systems indicated that in its present state the framework is definitely useful in practice. The transformation engine is available open-source:
`https://github.com/sed-szeged/cppbackport`

The paper is organized as follows. Section II presents more details on the practical scenario that lead to the development of the solution. Related work is briefly presented in Section III. Section IV describes the framework and its usage scenarios in detail, while the transformations themselves are listed in Section V. Section VI deals with the evaluation of the solution and our measurement results, together with Section VII, which lists the most important limitations of the approach, before the conclusion in Section VIII.

II. Motivation and Overview

iGO navigation software, the core product of NNG, is a *white label* product, meaning that clients can sell the final products under their own brand. Clients have significant freedom in customizing the user interface and application behavior to their taste, which produces high variability not only on the market, but on a technical level as well. While customizations have big impact on certain features and workflows, many core functionalities remain practically the same in the majority of the products. As a typical software product line [24], the iGO system has core assets that share a common code base, which has to compile in all supported environments.

In some segments, successful products have numerous new generations with newer and newer versions of the iGO core in them, but without significant changes in the hardware/OS layers. iGO core assets are required to support compilation environments for these legacy platforms as long as business interest [4] and support periods sustain the need. Two notable examples of such legacy target platforms are Windows CE and QNX 6.5. Windows CE can only be targeted with C++03 compilers, while for QNX 6.5 the compilation toolchain is based on GCC 4.4.2.

On one hand we see a clearly articulated C++03 compatibility requirement for several years. On the other hand C++11 and the more recent versions of the C++ language are not only minor refinements, but contain significant benefits over the legacy language. There are multiple aspects here. One group of them relates to product quality. Move semantics of C++11 allows faster code even without modifying the source code [22]. Many features of the new language help to enhance code expressiveness. Self-explanatory code without boilerplates is less error-prone and in turn leads to better quality and faster production.

The other key factor is developer retention/attraction. Not having major changes to C++98, in a few years we can refer to C++03 as a 20-years technology. Continuous learning is a vital part of the successful developer mindset [21]. Reliable extension and maintenance of a multiplatform C++ software product line requires skilled engineers, for whom modern C++ is the norm. Being forced to a 20-years technology with millions of lines of code in a non-trivial domain easily becomes a business issue because of this human factor.

The opposing business needs for the legacy and new C++ variants made NNG think in building a bridge between the two. The requirement is simple: be able to use as many of the modern C++ features in the common code base as possible without compromising compatibility with the still important legacy platforms.

Our first cooperation in this topic was a classic research project to come up with possible approaches and their detailed assessment for decision making. Table I contains the identified scenarios and their fitness from different angles. The three possibilities were: *Columbus*, a C++ analysis framework developed at the University of Szeged [9], the open-source *clang* front end for the LLVM infrastructure [18], and the *C backend* developed also for LLVM [17]. Each criterion was assessed on a scale of 1–5, as can be seen in the table. Finally, NNG decided to choose the *clang* code transformation approach, mostly because it is open-source while Columbus is not, and the C backend turned out to be incomplete and unreliable.

A. Overview of the solution

A high-level overview of the transformation process is depicted in Figure 1. Developers use a modern C++ IDE (e.g. Microsoft Visual Studio[2] 2015) in their daily job. Our tool generates the backported equivalent of the source tree, so when a legacy build or debugging is needed, legacy tools/IDEs (e.g. Microsoft Visual Studio 2005 or GCC 4.4.2) can be used naturally.

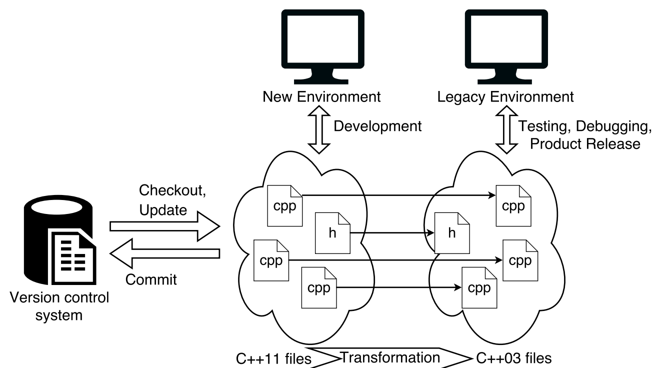

Figure 1. General use case of the framework

[2]https://www.visualstudio.com

Table I
POSSIBLE TRANSFORMATION SCENARIOS AND THEIR FITNESS (1 - BAD, 5 - GOOD) FROM DIFFERENT ANGLES.

Criterion	LLVM clang	Columbus	LLVM C back
Cost of development	2	2	1
Cost of integration into NNG processes	4	4	3
Learning curve	5	5	3
Degradation of work efficiency	3	3	1
Diagnostics	4	4	1
Performance: compilation	2	1	1
Performance: speed	4	4	1
Performance: memory	4	4	3
Performance: executable size	4	4	3
New language elements	1	1	4
Robustness	3	3	5
Future proof	1	1	3
Automation	5	5	5
Impact on iGO code	5	5	5
Support	3	4	1
Legacy compatibility	5	5	5

Apart from the transformation itself, our framework provides support for various every day software engineering activities such as testing and debugging. Since runtime issues (either from testing or operation phases) arise at the legacy production environment, while the developers should use their native development environment, the necessary traceability needs to be established on source code level.

For instance, bug reports of native systems may contain location references to the compiled executable. In case of a crash, for example, call stacks of different threads are dumped. This information together with a corresponding map file that matches the raw addresses to the source code are invaluable for finding the root cause of the bug. On legacy targets call stacks refer to the backported source code. For more seamless integration into the development processes, we have created a convenience tool that enables developers to lookup the source code location in the modern C++ source code even for addresses referring to the backported executable.

III. Related Work

This work deals with static code analysis for the purpose of source-to-source code transformation. The topic has a large literature, and there are many experimental and production tools developed for various languages, both free and commercial. Also, the application areas are diverse: language translation, (back)porting, modernization, refactoring, etc. In this section, we overview the common solutions for source transformation with special focus on the C++ language, and not particularly on the application of transformation.

Compiler infrastructures are often used for language translation, for instance the EDG front end [8], GNU GCC [10], the ROSE compiler infrastructure [25] and LLVM clang [18], which is the chosen platform for our tool as well.

There are solutions that not only offer a library for source transformation but a complete framework for this task. These frameworks often provide an own language to define the transformation and are easier to use being specific, though often bring higher overhead, more difficult learnability and less flexibility. For example, Lee et al. [16] created such an environment, which is highly flexible and can be extended with new languages as well. A similar system was offered by Bagge et al. [3] that provides support for source code instrumentation and optimization transformations, but this system supports only C++. There are additional experimental and commercial systems which could be possibly suitable for similar tasks, such as SrcML [6], TXL [27], ASF+SDF [2], Stratego [26], DMS toolkit [7], and several others.

We found that only LLVM provides a proper interface to its internal representation that is suitable for our purposes, so we are using this environment. A few additional applications based on the LLVM clang [18] front end are listed below. Clang Tools [5] is a toolset that includes a code transformation module as well. An interesting tool is modernizer, which transforms C++03 code to C++11, exactly the opposite of what we developed. This tool is appropriate for other tasks as well such as formatting and code style checking. Another application of this library is Include What You Use [11], with which the optimization of include files can be performed.

Transformation on C++ code for a different purpose was done by, for instance, Aigner et al. [1], which can be used to eliminate virtual function calls in C++ in order to improve the performance of the programs. Marangoni et al. [20] implemented a tool with which general C++ code can be automatically transformed to CUDA source code, which enables parallel execution of general C++ on video cards. Additional parellelization transformation tools have been implemented by Krzikalla et al. [15] and Magni et al. [19].

An interesting tool based on LLVM is C Backend [17], which is able to transform C++ code to C code. This could have potentially also been a solution to our problem (as most compilers still support C), however this system is still in a very experimental phase. The generated code is much slower than the original, furthermore it cannot handle a number of code constructs at all.

IV. Source Code Transformation Framework

The alteration of the source code is controlled by the transformation framework. It consist of two main parts: the first one is the engine providing incrementality, while the second one is responsible for performing the actual transformations. The incrementality engine monitors the code changes at file level and determines which files of the project need to be transformed (discussed in more detail in Section IV-B). Based on this list, the transformation engine performs the needed changes, which is the topic of Section IV-A.

During the design of the framework, it was an important requirement that the tool should be easy to integrate into the build processes; either as a pre-build step in traditional build systems or into continuous integration (CI) environments.

A. Source code transformation

For using the transformation framework, we have to know how the compilation units are compiled in their original build environment. We use the `compile_commands.json` file [14] for this purpose. This text file contains the necessary information, which is the following:

- **directory**: the working directory used during the build process. The following fields (command, file) are relative to this path.
- **command**: the command line used to compile the compilation unit.
- **file**: path of the compilation unit file.

This data has to be provided for each compilation unit. In Figure 2 we show an example `compile_commands.json` file content. If the project does not contain this file yet, then the user has to create one. The `compile_commands.json` file can be created automatically (with an external tool, like CMake) or manually. We did not prepare such a tool on our own, because the industrial partner did not require it.

```
[{
    "directory": "c:/work/projectDir",
    "command": "cl.exe -c Source1.cpp -o2",
    "file": "c:/work/projectDir/Source1.cpp"
}]
```

Figure 2. Content of a `compile_commands.json` file

Before the transformation starts, the framework copies the full project hierarchy into a work directory, which has to be provided by the user. The transformed code will be saved into this directory as well, so this code will be compilable with a C++03 compiler.

In the following, we will describe the transformation process. During designing the process it was important to take into consideration that there are also such new C++11 features which cannot be transformed in one step (e.g. lambda expressions nested into other lambdas), and some transformations depend on each other and have to be performed in more iterations in a predefined sequence.

The transformation process and its phases are shown in Figure 3. These are the following:

- The transformation tool expects the `compile_commands.json` file containing the project's compilation information as input.
- We maintain a database, which supports the incremental operation by storing the latest modification times and the dependencies between the source elements. During preprocessing, the transformation framework analyzes the dependencies between compilation units and selects those files which have to be transformed based on the database (see Section IV-B).
- It then iterates over the list of the transformations. (We will describe these in Section V.)
- After a transformation is done on all affected files, the framework saves the changes, and the incrementality engine updates the database with the file modification dates.

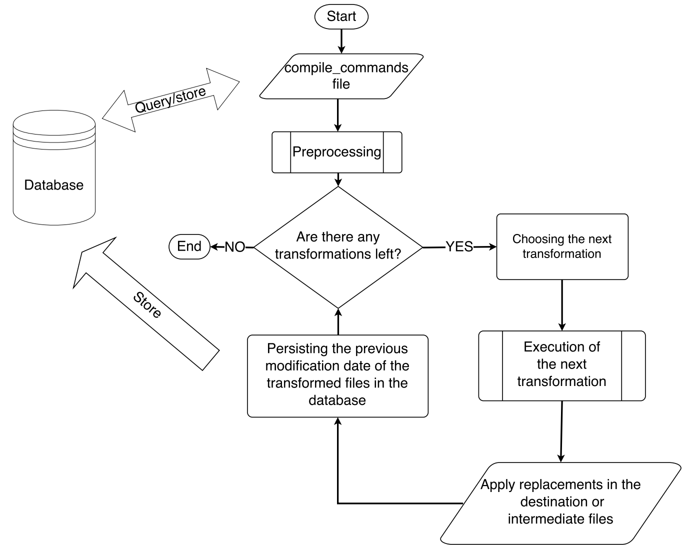

Figure 3. Flow chart of the transformation framework

B. Incrementality

It would take lots of resources to transform every file during each build of the project. This would be superfluous in most cases, because usually only a small fraction of the code gets changed during a development iteration. To eliminate this overhead, the framework records for each compilation unit file which version of it was already transformed, and it performs the transformation only if the file was modified in the intervening time. A file is considered to be modified if its last modification time changed. This is not the perfect solution, as the time attribute of a file can change even if its content does not, but this happens quite rarely and the side effect is not harmful.

Because we need to preserve the information between consecutive runs of the framework, we store the data in a persistent storage. We chose the SQLite[3] SQL-engine, because it does not need a database server and can be used easily without any configuration. However, the framework can be quickly adapted to other SQL engines (e.g. PostgreSQL, MySQL), if needed.

The database stores information about the compilation units (which are defined in the `compile_commands.json` file) and associated files for each unit. For each compilation unit, it stores the last modification date, the file dependencies (such as due to inclusion or given in command-line arguments), and the time stamp of the dependency addition. If a translation unit includes a header file, which also contains include-s, these dependencies will be added directly to the compilation unit,

[3]https://www.sqlite.org/

rather than to a dependency. (Dependent files cannot have dependencies this way.) Taking this into account, we developed the following simple database schema:

```
COMPILATION_UNIT(id, timestamp, cmd_args)
FILES(id, path)
RELATIONS(file_id, dep_id, dependency_timestamp)
```

C. Operation of the Transformation Framework

The framework collects the compilation units from the `compile_commands.json` file and by iterating over this list it collects also their dependencies (direct and indirect ones as well). Next, it compares this information with the database contents. If a compilation unit

- changed,
- its command line arguments changed,
- its dependencies changed,
- new dependency appeared, or
- existing dependency disappeared,

then the compilation unit gets inserted into the list of files to be transformed together with its dependencies.

This list will contain all files which might got modified since the last transformation, and the framework will perform the transformation of these files. If all transformations finish successfully, the framework updates the database by saving the new modification dates, adding possible new dependencies or deleting the disappearing ones. Furthermore, if new compilation units were added, these will also be added to the database together with their dependencies.

D. First analysis

Before starting the first analysis, the framework creates the database. If it already exists, it will not be overwritten. Next, the data tables will be created (if needed).

During the first run, the framework will transform all files, which can be time consuming in case of a larger project. Later however, because of the incrementality, only the changed files will be transformed.

E. Tracing the transformed code back to the original one

The traceability tool is a complementary tool for the transformation framework, which aims to create a mapping between the transformed and the original project, that is able to trace the lines between the original and the converted files. This is useful in cases when the transformed code contains an error, which, of course, has to be fixed in the original code. If it receives a transformed file and the line number in question, it returns the corresponding line in the original file.

Using the tool is limited in the sense that the back tracing can only be performed if it does not fall into a transformed region of code. If a line inside a transformed code part has been selected, it returns the back trace of the starting line of the outermost transformation. The reason for this restriction is that in case of some transformations the body of the transformed functions has to be written out with a procedure provided by clang. The problem is that while this code will be functionally the same as the original, it will differ in formatting. Perhaps the simplest example is that comments and blank lines are not printed out.

V. Transformation Catalog

In this section, we present the transformation details of the actual language elements supported by the framework. There are some other transformations available as well, which are in experimental phase and are mentioned in Section VII.

NNG's selection of which language elements to transform was based on their subjective usefullness/benefit judgement and the required efforts and complexity.

A. In-class data member initialization

The possibility to initialize class (union, struct) data members directly within their declaration in the class body has been introduced in C++11. This has the benefit that a data member which has a default value need not be initialized in each constructor but only once directly after its declaration. Earlier, this was possible only for data members with the `const static` modifier. The syntax for this construct is to use assignment operator or the brace initializer of the form `{ value }`. The construct has a restriction that only one member of unions can be initialized this way.

```
struct A {
  int a { 3 };
  std::string s = "s";


};

union B {
  double a = 3.5;
  int b;

};

class C {
public:
  C(int _b) : b(_b) {
  }
private:
  int a = 1;
  int b = 2;
};
```

⇒

```
struct A {
  int a;
  std::string s;
public: A() : a(3),
              s("s") {}
};

union B {
  double a;
  int b;
public: B() : a(3.5) {}
};

class C {
public:
  C(int _b) : b(_b), a(1) {
  }
private:
  int a;
  int b;
};
```

Figure 4. In-class member initialization examples

The listing in Figure 4 shows examples for in-class member initialization. The left-hand side of the figure lists the original C++11 code, and the other is the transformed version (C++03). The mechanism used for the transformation is practically the one used by the compiler as well. Namely, we move data initializers into the constructors provided they are not already present in the constructor initialization lists. Automatically generated constructors need special consideration. If they are not already generated by the front end, then our transformation framework will create them with public access specification (placed after the last existing member declaration in order not to accidentally modify visibility of other members).

Some member types are not handled by the framework because they cannot be transformed (or it is not practical) into its equivalent. This includes C-style arrays, because their members cannot be directly initialized in the constructor initializer lists, only in the constructor bodies by individual value

assignments. Also, declarations in which multiple declarators are provided for the same type are not handled. Finally, code is not transformed for template classes because in this case there might be constructors which are not instantiated by the front end, so consequently they could not be used to hold the generated code.

B. Auto type deduction

Prior to C++11, each variable (and other entity like a function return value) had to be explicitely declared for its static type. In many cases, this led to overly complex and unreadable code. The `auto` keyword used in place of a concrete type instructs the compiler to deduce the type of the entity automatically. However, in this case, the variable needs to be intialized at the declaration in order the type be deducible.

Our transformation framework uses the same deduction rules as the compiler but in our case, the source code with the deduced types is generated as well. In our implementation, various categories of auto types are distinguished, which is necessary because different treatments are required for the different cases:

- simple declarations
- multiple variables in one declaration
- function pointers
- template functions with such variables
- functions with trailing return types

```
auto a = 32;
auto *b = new auto(&a);
auto xp = &a, yp = xp;
auto *y = &a, **z = &y;
auto foo(int a)
      -> decltype(a) {
  return a;
}
auto x = foo(0);
const auto & y = foo(1);
auto fp = foo;
```

⇒

```
int a = 32;
int **b = new int *(&a);
int * xp = &a, * yp = xp;
int * y = &a, ** z = &y;
int foo(int a) {

  return a;
}
int x = foo(0);
const int & y = foo(1);
int (*fp)(int) = foo;
```

Figure 5. Auto type deduction examples

The listing in Figure 5 shows examples for auto type deductions with original and transformed code versions. This transformation has some limitations too. Namely, multiple variables for a declaration in global scope, template functions, and certain variable declarations combined with preprocessor macros are not fully handled.

C. Lambda functions

One of the most advanced new features in C++11 are lambda functions. With them, special functionalities may be written inline in a very compact way, without actually creating new functions each time, and which was possible only using function pointers or function objects in previous editions of C++. Our transformation engine translates lambda functions to function objects, as shown in the example in Figure 6.

```
std::vector<int> v(6);
int inc = 7;








std::for_each(
  v.begin(),
  v.end(),
  [&inc](int &n) {
    n += inc;
  }
);
```

⇒

```
std::vector<int> v(6);
int inc = 7;
class LambdaFunctor__12_1{
  int& inc;
public:
  LambdaFunctor__12_1(
    int& inc) : inc(inc) {}
  void operator()(int & n){
    n += inc;
  }
};
std::for_each()
  v.begin(),
  v.end(),
  (LambdaFunctor__12_1(inc))
);
```

Figure 6. Lambda function example

D. Attributes

The reason of the introduction of attributes in C++11 was to unify the creation of various compiler directives. Most compilers already implemented their dialect-specific ways for such directives, but this was not standard in any way (for example, construct like `__attribute__((...))` for GNU GCC and `__declspec()` for the Microsoft compiler). The use of attributes make this kind of extensions more portable, furthermore, they are very general and might be placed virtually at any syntactic position in the code, they might be placed in namespaces, can get parameters, etc.

```
[[attr1, attr2, attr3(args)]]
[[namespace::attr(args)]]
```

Figure 7. Attribute examples

Figure 7 shows what kind of attributes are accepted by our transformation framework. Since in the previous language versions there are no equivalent or similar code structures, we simply discard any occurrence of attributes from the code.

E. Final and override modifiers

The final and override modifiers were introduced to give developers compile-time control over class specialization and function overriding. These modifiers are not keywords in the language, and depending on the environment they can appear also as e.g. variable names. The `override` modifier indicates that the base class' virtual function is being overridden. The `final` modifier can be used with both virtual functions and classes. In case of a function it prohibits its overriding, while in case of a class it disables subclassing. The framework simply deletes these modifiers, similarly as in case of attributes. The listing in Figure 8 shows examples and their transformed versions.

```
class A {
  virtual void b();
  virtual void c() final;
};
class B final : public A {
  void b() override final;
};
```

⇒

```
class A {
  virtual void b();
  virtual void c();
};
class B: public A {
  void b();
};
```

Figure 8. Final and override modifier examples

F. Range-based for loop

In order to use the `for` loop easier in cases where an operation has to be performed on a whole range of elements, a more compact way of writing code was introduced. If the given container object has all the required special functions, it can be used in this simplified form. These special functions are called `begin` and `end`. An exception from this requirement are simple arrays, because in this case the range can be determined by calculating memory address offset. The special functions can be global or local. They are local if the two methods are defined in the class declaration and have no parameters, and global if they are defined outside the class in its enclosing namespace and have a parameter of the required class type.

```
int array[4]={1,2,3,0};


for (auto &k : array) {


  k = 1;
}
```

⇒

```
int array[4]={1,2,3,0};
int * __begin1 = (array);
int * __end1 = (array)+4;
for(;__begin1 != __end1;
      ++__begin1) {
  int &k = *__begin1;
  k = 1;
}
```

Figure 9. Range-based for loop example

During the transformation the new compact syntax is converted to the old form with three arguments as shown in the example code in Figure 9. Note that the introduced local variables are suffixed with a number to avoid name clash with further transformations in the same scope.

G. Constructor delegation

C++11 allows the delegation of constructors. This means that in the constructor initialization list another constructor can be called. In this case the constructor initialization list can contain only this single element. By using constructor delegation lots of copied code can be avoided when several constructors would perform similar initializations.

```
class A {
  A() {}
  A(string str) : s(str)
  {
    t = "hello";
  }
  A(string str, int dbl)
    : A(str) {

    a = dbl;
  }
  int a = 1;
  string s;
  string t;
};
```

⇒

```
class A {
  A() : a(1) {}
  A(string str) : s(str),
                  a(1) {
    t = "hello";
  }
  A(string str, int dbl)
      : a(1), s(str) {
    { t = "hello"; }
    a = dbl;
  }
  int a;
  string s;
  string t;
};
```

Figure 10. Constructor delegation example

The framework transforms the code in such a way that it copies the initialization list of the target constructor into the initialization list or body of the caller constructors, as can be seen in Figure 10. If the constructor delegation is used in template classes then the framework can transform only the instantiated constructors.

H. Type aliases

Supporting typedef-names is a long-standing feature of C and C++ to create aliases for existing types, but it does not support aliases which can receive template parameters. C++11 introduced a new syntax to support this feature with the `using` keyword. Using template parameters can come in handy in case of creating aliases for template classes. The listing in Figure 11 shows an example.

The framework converts the new syntax into the old format in simple non-template cases in a straightforward way. When there are template parameters, it creates a struct carrying the alias name and it inserts a typedef with the name 'type' into it. Also, all references to the alias are replaced by this construct. The listing in Figure 11 shows the transformed example code. Occurrences of the alias name in symbol import statements (`using` from base class, for example), and dependent names as alias parameters (requiring `typename` prefix for the nested type) are currently not supported.

```
using ul = unsigned long;
ul foo(ul p) {return p;}

template<class T>
using mapVec=std::map
  <T, Vec<T> >;



mapVec<int>
bar(mapVec<int> p) {
  return p;
}
```

⇒

```
typedef unsigned long ul;
ul foo(ul p) {return p;}

template<class T>
struct mapVec {
  typedef std::map
    <T, Vec<T> > type;
};

mapVec<int>::type
bar(mapVec<int>::type p) {
  return p;
}
```

Figure 11. Type alias examples

VI. Evaluation

We evaluated our transformation framework from two aspects: correctness of the transformed code and performance (runtime). The first aspect is clearly important since we want the transformed code be functionally equivalent to the original one. However, note that there are language constructs that are not handled by the framework, so these were excluded from our measurements (and were, of course, communicated to the users). We discuss functional testing in Section VI-A.

The second aspect of the evaluation, performance testing, is important since the framework is planned to be used in production by our industrial partner, integrated into the build process. Since the company employs frequent builds, which is resource intensive due to the large and complex code base, time to perform the transformation is also critical. Associated measurements are provided in Section VI-B.

During development and early stages of the evaluation, we used a set of code snippets with the language features of interest. Later we relied on a benchmark of systems, which use some of the C++11 features, and are non-trivial in size. We included two kinds of systems: four open-source systems and two proprietary ones. Some basic properties of the subject systems are provided in Table II. All subjects belong to different domains, and the sizes of the open-source systems range from small to medium, while the industrial ones can

be treated as large systems. The first industrial system is Columbus, our own source code analysis framework [9]. The other system is iGO, the product of our industrial partner NNG, which was the initial motivation for this work.

Table II
PROPERTIES OF THE SUBJECT SYSTEMS

	LOC	Transl. units	Transformations
SoDA[4]	18,849	126	193
log4cplus[5]	37,543	67	172
GridDB[6]	113,270	68	13
aria2[7]	118,063	385	3,388
Columbus	889,725	1,462	343
iGO	millions[8]	121	0

Lines of Code (LOC), given in the second column is counted as logical lines (not including empty and comment lines), while the number of translation units is essentially the number of source files with extension `.cpp`, that are compiled by the compiler during build. The last column of the table shows the number of transformations performed by the system during the whole process. It can be observed from the statistics that the actual number of transformed language elements varied from program to program and it did neither really correlate with program size nor with the number of translation units.

The reason behind the surprisingly low number of compilation units in the iGO system is a build time optimization technique called *unity build*. It works by processing a set of compilation units together so that multiple redundant processing of header files is radically reduced [23]. For iGO, there were no actual transformations performed, which is discussed in the following.

A. Functional testing

The correctness of the transformed code was checked in two steps. First, the transformation framework is capable of checking if the code is syntactically correct, so after each successful transformation this check was also performed. Second, the code has to produce the same behavior as the original one, and this property was verified at multiple levels:

1) We wrote a set of code snippets containing examples of the implemented transformations (see Table III for their amount). These pieces of test code have been transformed, syntactically checked, and compiled in the legacy environment. Then, each example was manually verified, and finally executed on one or two test cases for functional equivalence. These tests are part of the transformation framework available open-source.
2) On the four open-source systems and Columbus we also performed the transformation, syntax check, and legacy compilation. Finally, we manually verified a limited number of transformations performed in these systems (due to their large number, we could not check all).
3) In the case of iGO, there were no actual transformations performed, as can be observed from Table II as well. This is because at the time of the experiments the code base did not include any C++11 features. However, the other parts of the process – analysis, compilation, integration into the build process, incrementality, etc. – were verified. To check the actual working of the transformation engine, the code was temporarily modified at a few places to include C++11 code.

[4]https://github.com/sed-szeged/soda
[5]https://github.com/log4cplus/log4cplus
[6]https://github.com/griddb/griddb_nosql
[7]https://github.com/aria2/aria2
[8]the exact figure is confidential

Despite the fact that no actual transformation has been done on iGO yet, the above functional testing process ensures future usability of the framework on this system as well. The transformed code needed to be platform independent, so we performed the tests on Windows and Linux environments with different compiler versions as well.

Table III
CODE SNIPPETS FOR FUNCTIONAL TESTING

Transformation	Code snippets
In-class data member initialization	3
Auto type deduction	37
Lambda functions	31
Attributes	3
Final and override modifiers	3
Range-based for loop	9
Constructor delegation	2
Type aliases	3
All	91

B. Performance testing

In order to improve the applicability of our framework on big systems we implemented different speedup techniques to reduce the overall processing time:

- Transformation is running in multiple threads in *parallel*.
- *Incremental* transformation performs only the necessary steps based on what has changed since the last transformation.
- *Feature finder* identifies what language features are used in the different compilation units to eliminate their superfluous processing in the unrelated transformation rounds.
- The *MultipleTransforms* phase performs transformations of certain independent language features in a single round.

The following discussion presents measured processing times and other empirical results on our reference code bases.

Figure 12 shows total processing times on the code bases of the four open-source systems in seconds.[9] The same performance test was performed with different parallelization settings (how many threads to use) to determine the scalability

[9]Source code was accessed via a mapped network drive, presumably resulting in slower than usual file access times, somewhat distorting the measurements.

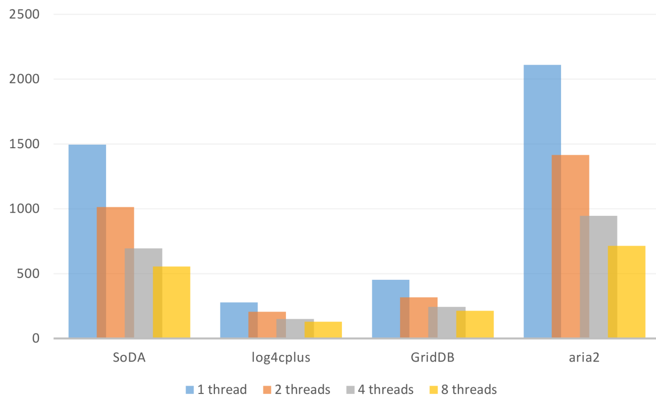

Figure 12. Runtime in seconds.

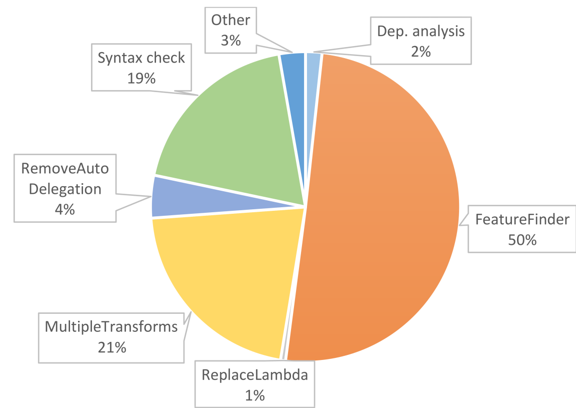

Figure 13. Average distribution of time spent in transformation phases

of the framework. Note that *GridDB* has a big advantage in terms of translation time compared to *SoDA*, even though the LOC measure of the former is 6 times of that of the latter. The big difference is caused by not including 3rd party code when counting LOC, while the transformation framework has to analyze 3rd party code as well. Systems may have certain large 3rd party codes embedded into their own code base, resulting in lots of extra instructions to process by the transformation framework. Currently the last phase, when syntax check is performed, does not support parallel execution, which reduces scalability to multiple cores.

Table IV presents processing times by phases without parallelization.[10] The transformation starts with *Dependency analysis* which checks each compilation unit and its dependencies and decides whether the compilation unit has to be transformed or not. The most time consuming phase is clearly shown to be *FeatureFinder*, being responsible for identifying language feature usages, because it has to examine all compilation units. The transformation phases (*ReplaceLambda*, *MultipleTransform* and *RemoveAutoDelegation*) and Syntax check phase (which verifies the transformed code) only deal with units containing code fragments relevant to the actual transformation phase. The big differences between times of *FeatureFinder* and the certain transformation phases reveal how much time is saved by the feature finder optimisation. Though transformation phases do not only parse source code, but also transform it, time spent in actual code transformations was measured to be negligible compared to parsing time.

The distribution of the four open-source systems' processing time among the different transformation phases is shown in Figure 13. The numbers were determined by averaging values of Table IV. 71% of the time is spent in the *FeatureFinder* and *MultipleTransforms* phases. Without *FeatureFinder* the distribution would probably be more equalized, since each phase would contain very similar parsing and the negligible code transformation steps for the same complete set of compilation units. *MultipleTransforms* eliminates entire transformation phases by uniting the processing of independent language features.

[10] Although iGO did not contain any C++11 features to transform, the other phases of the process were executed.

VII. Limitations

Apart from the ones listed in Section V, our framework implements several other transformations, though with limited functionality. This includes the following language features: variadic templates, rvalue references, move constructors and `decltype` specifiers used for type deduction. These features can be used provided some constraints are met by the developers, but since the most typical usage scenarios are handled, this does not mean serious limitation in practice.

In Section V, we already listed some concrete limitations for the transformations (e.g., unused template methods, deletion of attributes). Apart from these, if the framework encounters some specific variants of language features that are not fully handled, it tries to skip those parts and continue the analysis, before eventually terminating with an error. If the system contains code that is generated during compilation, the framework will not consider these files.

Fully automatic generation of the `compile_commands.json` file required for building with the clang infrastructure is not supported. In Linux, the CMake[11] system provides functionality for generating this file, while on other systems Bear[12] might be used. However, some additional modifications are needed to be made on the generated file in order to be compatible with the transformation framework. As far as we know, for Windows systems there is no universal solution for producing the build file, so in this case the user has to provide it. A particular issue on Windows is related to older Visual Studio versions,[13] in which case the project file has to be prepared (or updated) in multiple versions, one for each Visual Studio edition.

Finally, each subject system to be transformed needs to be compilable by the clang compiler, because this is what our framework is built on. Systems not satisfying this property might require significant porting effort before being capable of transformation.

[11] https://cmake.org
[12] https://github.com/rizsotto/Bear
[13] https://msdn.microsoft.com/en-us/library/ms950416.aspx

Table IV
DETAILED RUNTIME DATA WITHOUT PARALLELIZATION (IN SECONDS)

	Dep. analysis	FeatureFinder	ReplaceLambda	MultipleTransforms	RemoveAutoDelegation	Syntax check	Total
SoDA	47	853	3	281	35	239	1,458
log4cplus	3	136	3	69	12	48	271
GridDB	4	285	0	57	0	103	449
aria2	20	860	16	508	222	333	1,959
Columbus	142	4,631	73	2,672	617	1,345	9,480
iGO	202	2,319	N/A	N/A	N/A	905	3,426

VIII. CONCLUSION

There are many reasons why companies are facing problems when they need to produce C++03 code but their developers are eager to use the new features of C++11. This motivated our work to construct a system for automatically transforming C++11 code to C++03. The system allows, under certain restrictions, for developers to use various C++11 language elements so that after conversion, software will continue to be compatible with the older C++03 standard. We designed the system in a way that it can be easily integrated into a wide range of development processes. In addition, it provides several other services, such as incremental transformation, cloning source code structure, and source traceability.

We detailed the features and capabilities of our source to source transformation system, which includes the basic structure and operation of the framework, the implemented transformations with examples, and information on how we tested them. We evaluated system performance on different open-source applications and on two large industrial systems, highlighting the scalability and some limitations we encountered. We know that the testing methodology we used for validation could be enhanced further, but current experience shows that the method is already usable in practice.

The developed framework is open-source and it can be freely used. There are many opportunities for further development, however. For instance, handling new language elements, correction of current transformation errors, and the improvement of error recovery mechanisms.

ACKNOWLEDGMENT

We are grateful to our industrial partner for the interesting project. The authors would like to thank Ádám Maróti, Ádám Rudas and Károly Szabó for their work in the implementation. This work was partially supported by the European Union FP7 project REPARA (no: 609666).

REFERENCES

[1] Gerald Aigner and Urs Hölzle. *ECOOP '96 — Object-Oriented Programming: 10th European Conference Linz, Austria, July 8–12, 1996 Proceedings*, chapter Eliminating virtual function calls in C++ programs, pages 142–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[2] ASF+SDF meta-environment. http://www.meta-environment.org/. Accessed: 2016-06-21.

[3] Otto Skrove Bagge, Magne Haveraaen, and Eelco Visser. CodeBoost: A framework for the transformation of C++ programs. Technical report, 2001.

[4] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, Jan 1995.

[5] Clang Tools. https://github.com/llvm-mirror/clang-tools-extra. Accessed: 2016-04-17.

[6] Michael L Collard, Jonathan I Maletic, and Brian P Robinson. A lightweight transformational approach to support large scale adaptive changes. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[7] DMS software reengineering toolkit by Semantic Designs. https://www.semanticdesigns.com/Products/DMS/DMSToolkit.html.

[8] EDG C++ Front End. https://www.edg.com/c.

[9] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, October 2002.

[10] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Accessed: 2016-04-24.

[11] Include What You Use. http://include-what-you-use.org/.

[12] ISO/IEC 14882:2003 - Programming languages – C++. http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110. Accessed: 2016-04-23.

[13] ISO/IEC 14882:2011 - Information technology – Programming languages – C++. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372. Accessed: 2016-04-23.

[14] JSON Compilation Database Format Specification. http://clang.llvm.org/docs/JSONCompilationDatabase.html. Accessed: 2016-04-23.

[15] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. *Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29 – September 2, 2011, Revised Selected Papers, Part II*, chapter Scout: A Source-to-Source Transformator for SIMD-Optimizations. Springer Berlin Heidelberg, 2012.

[16] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. *Languages and Compilers for Parallel Computing (LCPC 2003), TX, USA, October 2-4, 2003. Revised Papers*, chapter Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation. Springer Berlin Heidelberg, 2004.

[17] LLVM C Backend. https://github.com/draperlaboratory/llvm-cbe.

[18] LLVM clang compiler infrastructure. http://clang.llvm.org. Accessed: 2016-04-17.

[19] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 11:1–11:11, New York, NY, USA, 2013. ACM.

[20] Marangoni, Matthew, Wischgoll, and Thomas. Paper: Togpu: Automatic Source Transformation from C++ to CUDA using Clang/LLVM. *Electronic Imaging*, 2016(1):1–9, 2016-02-14T00:00:00.

[21] Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011.

[22] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[23] József Mihalicza. *Analysis and Methods for Supporting Generative Metaprogramming in Large Scale C++ Projects*. PhD thesis, Eötvös Loránd University, 2014.

[24] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[25] ROSE compiler infrastructure. http://rosecompiler.org/. Accessed: 2016-06-21.

[26] Stratego/XT program transformation language. http://strategoxt.org/. Accessed: 2016-06-21.

[27] The TXL programming language. http://www.txl.ca/. Accessed: 2016-06-21.