Procedia Computer Science
Volume 80, 2016, Pages 1485–1496
ICCS 2016. The International Conference on Computational Science

Source Transformation of C++ Codes for Compatibility with Operator Overloading

Alexander Hück[1], Jean Utke[2], and Christian Bischof[1]

[1] Institute of Scientific Computing, Technische Universität Darmstadt, Germany
{alexander.hueck, christian.bischof}@sc.tu-darmstadt.de
[2] Quantitative Research & Analytics, Allstate Insurance Company, USA
jutke@allstate.com

Abstract
In C++, new features and semantics can be added to an existing software package without sweeping code changes by introducing a user-defined type using operator overloading. This approach is used, for example, to add capabilities such as algorithmic differentiation. However, the introduction of operator overloading can cause a multitude of compilation errors. In a previous paper, we identified code constructs that cause a violation of the C++ language standard after a type change, and a tool called OO-Lint based on the Clang compiler that identifies these code constructs with lint-like messages. In this paper, we present an extension of this work that automatically transforms such problematic code constructs in order to make an existing code base compatible with a semantic augmentation through operator overloading. We applied our tool to the CFD software OpenFOAM and detected and transformed 23 instances of problematic code constructs in 160,000 lines of code. A significant amount of these root causes are included up to 425 times in other files causing a tremendous compiler error amplification. In addition, we show the significance of our work with a case study of the evolution of the ice flow modeling software ISSM, comparing a recent version which was manually type changed with a legacy version. The recent version shows no signs of problematic code constructs. In contrast, our tool detected and transformed a remarkable amount of issues in the legacy version that previously had to be manually located and fixed.

Keywords: C++, Type Change, Static Analysis, Source Transformation, Operator Overloading, Algorithmic Differentiation

1 Introduction

Scientific simulation codes and their underlying mathematical models are steadily evolving with respect to new insights and requirements. At the same time, these models need to be verified and analyzed regarding their modeling fidelity. Consequently, the ability of the software to provide derivatives is necessary for many approaches such as stability and sensitivity analysis

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016
© The Authors. Published by Elsevier B.V.
doi:10.1016/j.procs.2016.05.470

[3] or providing accuracy information of the numerics of the simulation [21]. Here, algorithmic differentiation (AD) [8] can deliver derivatives accurate up to machine precision at a lower computational cost compared to black-box methods. In the context of C++, this kind of semantic augmentation can be achieved by using operator overloading. To that end, a new user-defined type $\widetilde{T}$ replaces the original type T (e.g., `double`) for the computations. $\widetilde{T}$ is a class that provides the same set of operations as the original type, e.g., arithmetic operators and functions. We deal with new types that use complete encapsulation [6], i.e., T is associated with $\widetilde{T}$ which encapsulates the old value in some way. In a similar fashion, capabilities such as interval arithmetic [1] can be inserted.

Modern simulation software can be written with these extensions in mind as the C++ language provides the necessary features to do so [23]. Often, legacy software was not written with these extensions taken in mind. Then, an extensional feature addition can be considered as adaptive software maintenance [10]. Examples for the addition of new semantics using operator overloading is the aforementioned ability to compute derivatives for complex and mature CFD software [27, 31]. However, operator overloading does not always work without refactoring the target source code in order to make it compatible with the user-defined type [25].

In a previous paper [9], we showed how the introduction of a user-defined type can break the code when using operator overloading. This is due to the difference between the treatment of built-in and user-defined types in the C++ language standard [11]. Our experience is based on type changes that were applied to large scientific code bases such as the CFD software package OpenFOAM (OpenField Operation and Manipulation, [12]) and ISSM (Ice Sheet System Model, [16]). Here, a plethora of errors is caused by the type change which necessitates refactoring of code at multiple locations.

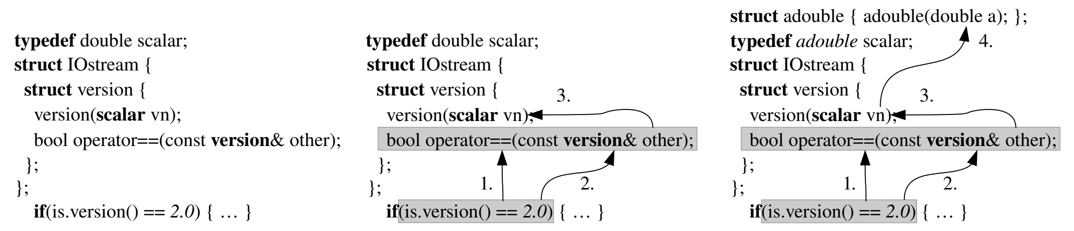
We developed a tool providing automatic source to source transformation to support developers of legacy software during the maintenance process. It takes the source files of the scientific code as input, finds problematic code constructs that can cause errors when a user-defined type is introduced, and automatically resolves them.

This paper is structured as follows: In Section 2 we discuss related work in the context of static source code analysis and transformations. In Section 3, we discuss problematic code constructs, that, after a type change has been applied, cause a violation of the C++ standard. Here, we also consider a new issue, the problems caused by potentially ambiguous name lookup. In Section 4, we describe how we identify these problematic code construct using Clang's matching abilities, and describe how we take care of them. The next two sections present empirical studies of OpenFOAM and ISSM, respectively. As it turns out, showing that the problematic code constructs do in fact appear with regularity.

2 Related Work

The idea of flagging problematic code constructs (including coding guidelines and style) and potential bugs started early for the programming language C [13]. Since then, the term "lint-like" became synonymous for statics code analysis tools that report potential problems to the users and might suggest a *fix-it* hint. These lint-like tools exist for most major programming languages, e.g., C++ [4, 19, 30], sometimes integrated in an IDE [7].

Static code analysis tools are often combined with source transformation features. This can be done to automatically fix potential bugs, style errors or rejuvate legacy codes by replacing deprecated language features [15]. As an example, the tool Clang-tidy [4] supports the transformation of certain C++ statements to modern C++11 features, e.g., introducing the keyword `auto` or transforming raw pointers to smart pointers.



```
typedef double scalar;
struct IOstream {
  struct version {
    version(scalar vn);
    bool operator==(const version& other);
  };
};
    if(is.version() == 2.0) { … }
```

(a) Example code

```
typedef double scalar;
struct IOstream {
  struct version {
    version(scalar vn);
    bool operator==(const version& other);
  };
};
    if(is.version() == 2.0) { … }
```

(b) Implicit conversion

```
struct adouble { adouble(double a); };
typedef adouble scalar;
struct IOstream {
  struct version {
    version(scalar vn);
    bool operator==(const version& other);
  };
};
    if(is.version() == 2.0) { … }
```

(c) User-defined type: scalar

Figure 1: (a) Pseudo C++ example code using the alias `scalar`. (b) A conversion (`2.0` to a `version` object) is implicitly introduced by the compiler due to the comparison operation. (c) The introduction of a new type $\widetilde{T} \equiv \mathit{adouble}$ breaks the code as there are two user-defined conversions (`2.0` to an `adouble` object to a `version` object) which is illegal according to the C++ standard ([11], §12.3-4).

There are many frameworks and tools supporting the structured rewriting and transformation of source code. ROSE [24] is a popular compiler framework supporting the source to source transformation of C/C++. Tools based on ROSE modify the abstract syntax tree (AST) nodes directly, which subsequently can be unparsed, thus resulting in transformed code. One application is a tool for AD that augments codes with derivative statements [22]. Similarly, the Clang compiler framework [18] supports user-defined analyses on the AST, but unlike ROSE, its AST is considered to be immutable. Thus, source transformation operations are modifying the source text directly. Clang is used for the loop vectorization of C codes [14], transformation from CUDA to OpenCL source codes [20] and several static source analysis and transformation tools, e.g., [19, 29].

3 Caveats of Operator Overloading

The typical situation is that, with built-in types T, the code will compile but the introduction of a user-defined type $\widetilde{T}$ is causing compile time errors with certain code constructs. An example of this scenario is shown in Figure 1. The following issues were discussed in detail in [9]:

Implicit Conversion is a conversion from type T to another type U without an explicit conversion statement in the code (cf. Figure 1a). These conversions are automatically added by the compiler. For $\widetilde{T}$, the standard restricts [11] §12.3-4: "At most one user-defined conversion [...] is implicitly applied to a single value."

Implicit Boolean Conversion is a subset of implicit conversions. Here a type T is transformed to a Boolean type, e.g., in an if-condition `if(a)`. The compiler may transform many built-in types, e.g., floating-point and integral types. $\widetilde{T}$ needs to implement a bool conversion function to support this ([11], §12.3.2). However, this conversion function and operator overloading should not be mixed due to undesired side-effects as stated in [26].

Explicit Conversion is a conversion from type T to another type U with an explicit statement in the code ([11], §5.4). Here, the encapsulated value T needs to be exposed for the conversion of $\widetilde{T}$ to work.

Unions are a type of class where members of the union share the same memory region. $\widetilde{T}$ "with a non-trivial constructor, [...] non-trivial copy assignment operator cannot be a member of a union" ([11], §9.5). The C++11 standard relaxes this restriction but forces the developer to explicitly provide the set of non-trivial functions of a member for the union by deleting these functions [26].

Conditional Assignment is used to set a value of some variable based on a condition. In C++, this can be achieved by the conditional operator expression `?:` ([11], §5.16) or an equivalent `if-else` statement. For some AD tools based on operator overloading, the internal bookkeeping may require replacing these structures.

Scope Resolution Operator and Friend Functions: Friend functions are used to access private members of a class but they are themselves not members (not invoked on an object) and are not in the scope of the class. The scope resolution operator can be used for a qualified lookup, specifying the scope of a declaration. Qualified lookups (e.g., `::sqrt(...)`) disable argument dependent lookup (ADL, [11] §3.4.2) which is used by the compiler to find friend functions that lack declarations in the scope of the respective class. Thus, a compile time error is caused. This, however, pertains mostly to the design of $\widetilde{T}$.

Name Lookup is a newly discovered cause of errors and pertains to finding the corresponding declaration of a name (e.g., a function call encountered by the compiler) ([11], §3.4). The lookup can be unqualified or qualified using the scope resolution operator.

For maximal compatibility with a code base, the user-defined type has to provide overloaded operators as well as overloaded functions, e.g., `sqrt`. To that end, the class and corresponding overloaded (mathematical and arithmetical) functions may reside in the global namespace. Thus, calls and operations w.r.t. $\widetilde{T}$ behave as if it were a plain floating-point type.

If a software defines its own implementations of, say, a mathematical function in its own namespace, a lookup might become ambiguous if $\widetilde{T}$ provides a corresponding overload. The compiler can not decide which declaration to choose, as exemplified in Listing 1.

```
T̃ sqrt(const T̃& a);
namespace ns {
  typedef T̃ scalar;
  scalar sqrt(scalar v);
  scalar a;
  ::sqrt(a); // OK: T̃ sqrt(const T̃& a)
  ns::sqrt(a); // OK: sqrt(scalar v), same as ::ns::sqrt(a)
  sqrt(a); // Error: Ambiguous call
}
```

Listing 1: Example of potential calls in a code to a function (`sqrt`). If the user-defined type provides an overload, the last call is ambiguous as the compiler finds two matching declarations.

This ambiguity can be resolved in multiple ways depending on several factors. In the following, we present two possibilities and discuss their implications regarding non-templated code. We refer to any function that is overloaded by the type $\widetilde{T}$ as well as present in the inner namespace `ns` as φ. It takes as argument types either T or $\widetilde{T}$.

One solution is the removal of all definitions of φ in `ns::`. Consequently, the qualified lookups (`ns::` or `::ns::`) must be dealt with accordingly. Thus, all these function invocations have to be transformed to unqualified lookups, e.g., `ns::sqrt` to `sqrt`. This will cause the compiler to chose the overloaded function in the global namespace. A caveat of this solution are calls to φ from within a method declaration which belongs to a class that also defines a method with the same name as φ. Thus, for every instance of a call to φ inside a method of such a class, the call has to be changed to a qualified lookup in the global namespace, i.e., `::`φ, where the overloaded function resides.

Another approach is the full qualification for each call to φ. It is preferable as it avoids the ambiguity described above. The full qualification requires precise knowledge about the location of the callee, i.e. in which namespace the name resides. Here, the callee has to be (made) compatible with $\widetilde{T}$.

Automatically resolving this issue is in general not possible without knowing more about the design principles underlying these namespace qualifications.

4 Transforming C++ Source Codes

Our focus is to prepare programs for semantic augmentation. Here, we limit ourself to code constructs that are generally problematic for user-defined types using operator overloading. As such, we exclude conditional assignments and scope resolution operator/friend function usage as they are dependent on the design of the user-defined types.

Our transformations target source codes before the semantic augmentation is applied. The goal is to change as little of the syntax of the original code as possible, i.e., apply the least code changes with the least possible side-effects to enable operator overloading compatible with all relevant C++ standards. For refactoring or code renovation this is often a requirement [28]. To that end, we analyze the AST of the respective translation unit, i.e., a (`.cpp`) file of the project from which an object file is generated. We employ the Clang compiler framework [18] for this task as it is a production level C++ compiler, providing an infrastructure to write tools that work on syntactic and semantic information of a program. Specifically, we employ the `libTooling`[1] library which handles the creation of the AST, including semantic nodes that are necessary for a meaningful static analysis. Additionally, Clang offers techniques for sophisticated pattern matching on the AST structure as well as facilities to apply source transformations.

4.1 Utilizing the Clang Compiler Framework

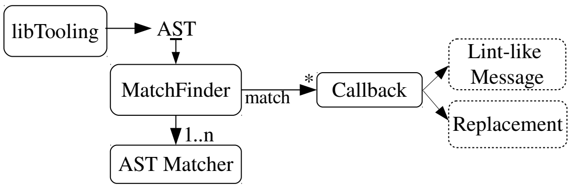
We are mostly concerned with two main phases of Clang's AST creation: The preprocessor phase where includes are handled and the final, semantically annotated AST which contains the expanded macros. We utilize the callbacks of the preprocessor to determine the correct locations of `#include` statements that are added during certain transformations. The final AST is used for the static analysis and source transformation operations. Figure 2 shows the principle behind searching the AST and subsequent transformations of nodes in the AST. In the sequel, we describe some properties of the AST and pattern matching for the static analysis before describing the application of source transformations. Generally, template code and transformation of macro definitions are excluded as Clang does not retain alias information for template instantiations and the transformation of macro definitions may have side effects.

4.1.1 AST and Pattern Matching

The Clang AST carries all necessary information, including implicit expressions that are added as dedicated nodes during the semantic analysis together with typedef information. Hence,

[1]http://clang.llvm.org/docs/LibTooling.html

Figure 2: Node matching: A match is found with AST matcher expressions. A callback for each node is invoked. Replacements describe the textual changes that need to be applied to a source code file.



for an expression it is possible to detect if a specific typedef alias is present. We exploit this additional information to make our analysis more accurate. Instead of searching for all implicit conversions, explicit casts etc. with a plain double data type, we limit our search to nodes with the respective typedef alias. As an example, Figure 3a represents the slightly simplified AST for the if statement of our example code in Figure 1a.

```
IfStmt
└─CXXOperatorCallExpr '_Bool'
  ├─...
  └─MaterializeTemporaryExpr 'Field::version'
    └─CXXConstructExpr 'Field::version' 'void (scalar)'
      └─FloatingLiteral 'double' 2.000000e+00
```

(a) Simplified Clang AST

```
StatementMatcher impl_conversion =
    materializeTemporaryExpr(
      hasTemporary(ignoringImpCasts(
        constructExpr(
          hasImplicitConversion(type_s),
          unless(temporaryObjectExpr())
        ).bind("conversion")
```

(b) AST Matcher for `type_s`, e.g., "scalar"

Figure 3: (a) Simplified Clang AST focusing on the right hand side expressions of the comparison (`==`) operator call shown in Figure 1. A temporary object is created by passing the literal to the version class constructor. (b) The AST matcher finds such instances.

We employ so-called AST matchers which are a query language for AST nodes. Expressions can be assembled in order to find relevant nodes. Figure 3b shows a matcher expression to find nodes that represent implicit conversions. To that end, we match the node `MaterializeTemporaryExpr` which binds a temporary (ignoring all implicit casts) and has the child node `CXXConstructExpr` which represents a call to a constructor. We only search for objects that are created from a single explicit argument (not counting default arguments of a constructor [11], §8.3.6) with a mismatching argument and class constructor type. In the AST we have shown that a temporary version object from a literal is created. The struct `version` in Figure 1a expects a `scalar` type, which represents a mismatch with the literal and causes a subsequent compilation error after a type change. We check for this by employing our own matcher extension `hasImplicitConversion`. We ignore nodes of the subtype `TemporaryObjectExpr` as they represent explicit functional casts instead of implicit constructor calls.

4.1.2 Source Transformation

The source transformation in Clang is based on string manipulations as the AST is immutable. To that end, there is a tight coupling between the AST nodes to the actual textual representation of the source code. The location of implicit nodes points to the respective target expression. Source transformation operations are by `Replacement` objects, containing positional information and the replacement string. As such, the transformed source code file is unchanged except for the `Replacement` target strings. We do not yet handle transformations on macro code. It is not safe to transform them as potentially unrelated code sections also make use of said macro and could, thus, be affected. In the following, we briefly describe our source transformations for each of the relevant cases. We give an example of a problematic code construct on the left and the transformation result on the right. Here, the variable a stands for type scalar $\equiv \widetilde{T}$.

Implicit Conversions (cf. Figure 1) are handled by adding a functional cast to the expression causing it, thus, removing one implicit conversion step.

```
if(is.version() == 2.0) { }
```

```
if(is.version() == scalar(2.0)) { }
```

Implicit Bool Conversions are resolved by adding an explicit comparison operation.

```
if(a) { }
```

```
if(a != scalar(0.0)) { }
```

Explicit Conversions are replaced by a template function reCast that was proposed in a previous paper [9]. Alternatively, the C++11 standard allows for the definition of conversion operators which have to be explicitly invoked using the keyword explicit [26]. Thus, for every explicit conversion in the code, the compiler will use such a user-defined conversion operator. However, this is an option only if the user-defined type supports this feature and the target code of the type change supports the compilation with C++11 features. The proposed reCast function can be specialized for the type $\widetilde{T}$ and, thus, can be employed to support a wide range of user-defined types independent of a C++11 explicit conversion operator implementation.

```

int b = static_cast<int>(a);
```

```
#include <recast.h>
int b = reCast<int, scalar>(a);
```

Unions can be named or anonymous. A named union is transformed to a struct. An anonymous union is simply removed, retaining its members in the same namespace as the union. In both cases, this retains the access pattern. However, the inherent memory overlay is lost and the memory consumption will rise by a fixed amount relative to the previous consumption of the union. Furthermore, the initialization of a union is handled differently compared to a struct, plus the assignment to one of the transformed members does not influence the other members as would be the case for a union. If such assumption regarding the access were made, our transformation will break the code.

```
union X { scalar a; int b; } // named
union { scalar c; int d; }
```

```
struct X { scalar a; int b; }
scalar c; int d;
```

4.2 Overhead of the Static Analysis Tool

We briefly compare the overhead of the static analysis compared to the standard compilation process. We report on the wall clock time of the compilation and analysis of part of the OpenFOAM code base, consisting of 431 translation units (cf. Section 5). The measurements were taken on a Linux-based single socket workstation, equipped with an Intel Xeon E3-1245 v3 (Haswell) with four physical cores, a Samsung 840 Pro solid state drive and 16 GB DDR3 RAM. The Clang compiler framework in version 3.5 is employed.

Similarly to a build system spawning multiple jobs to compile in parallel, our tool works on each translation unit separately and, thus, allows for a parallel analysis of the target software. In [29], the authors also exploit this to allow for static analysis and source transformation across many computers.

The static analysis tool takes 263.76 s using four concurrent processes. In contrast, the compilation process takes 77.14 s with four active jobs. This represents an induced overhead of about 3.4. At the file level, we picked a translation unit that was one of the slowest to analyze at approximately 6.31 s. In comparison, the Clang compiler took 1.33 s to produce the respective object file.

5 Empirical Study of OpenFOAM

We analyzed the CFD software package OpenFOAM `2.4.X`. It has a large C++ source code base with more than 700,000 lines of code (LOC).[2] It makes heavy use of templates, preprocessor macro definitions and other object oriented techniques throughout the whole code base. As such, it is a challenging target for a type change. In the sequel, we present our results of applying our analysis and transformation tool on the `src/OpenFOAM` source folder which totals more than 160,000 LOC. It represents the core of the project, defining the data structures and other fundamental features used by the provided solvers.

In total, we analyzed 431 C++ files. Our analysis is focused on resolving generally applicable problems with user-defined types. To that end, we have created a user-defined class with the goal to be maximally compatible with operations pertaining to the built-in `double` type. It overloads all necessary arithmetic, comparison and assignment operators as well as many mathematical and arithmetic functions. Using this simple user-defined type, we can abstract away any problem arising from the design of any specific user-defined type and concentrate on the inherent problems. The type change is done by changing the type of the OpenFOAM specific typedef alias `doubleScalar`. It is used to define the alias `scalar` which is used throughout the OpenFOAM code base as the fundamental data type.

5.1 Results

In total, we searched for the four different code constructs that were previously described in Section 3 and automatically fixed them in the source code. Table 1 shows the result of our findings. The first column identifies the file where the problematic code construct was identified in. The *# Includes* indicate how many times the file was included elsewhere, a count of one refers to the file itself. The results include a search for the `scalar` and `doubleScalar` alias.

			Applied Module			
File	**# Includes**	**Count**	Implicit Conversion	Boolean Conversion	Explicit Conversion	Union
IOstream.H	425	3			3	
token.H[×]	421	1				1
Field.C	303	1	1			
indexedOctree.C	7	3			3	
dynamicIndexedOctree.C	2	1			1	
SVD.C	1	1		1		
TimeIO.C	1	4			4	
Time.C	1	7			7	
outputFilterOutputControl.C	1	2			2	

[×] Detected searching for the alias `doubleScalar` instead of `scalar`

Table 1: Unique source of all statically resolved problems in OpenFOAM 2.4.

[2]Measured with `cloc` [5] on the project root `src` folder

After the source transformation, all errors pertaining to our identified code constructs disappeared. However, more errors are revealed, unrelated to the five transformed problematic code constructs. The source of errors are ambiguous name lookups as described in Section 3. OpenFOAMs implementation of these functions is trivial, simply calling the function in the global namespace. An example is shown in Listing 2.

```
namespace Foam {
  scalar sqrt(scalar v) { return ::sqrt(v); }
}
```

Listing 2: Mathematical function definition in the Foam namespace.

We wrote a prototypical matcher to find function calls involving the `scalar` type related to this problem. Our preliminary analysis of the OpenFOAM folder identified more than 150 distinct calls that either are (fully) qualified w.r.t. the Foam namespace, thus making it impossible to remove the OpenFOAM function definition, or unqualified calls leading to the observed ambiguous invocations. Despite these remaining challenges, we have successfully applied our source transformation tool and automatically fixed several problematic source code locations that caused errors with user-defined types. These code modifications are valid with either a built-in or user-defined type.

6 Empirical Study of ISSM

A second use case is presented for the Ice Sheet System Model (ISSM). It is an ice flow modeling software developed by NASA/JPL and UC Irvine and used by cryosphere scientists to project the future evolution of polar ice caps such as Greenland or Antarctica. Given the application area, the main goal, unsurprisingly, is the computation of large scale gradients for sensitivity studies, data assimilation, and other related uses. ISSM is implemented as a relatively large scale C++ code, however, with data (pre- and post-) processing done in Matlab. Noteworthy characteristics of the implementation are a modularization partially achieved by build-time and partially by run-time configuration, parallelization based on MPI (Message Passing Interface), the use of external solver libraries, and integration of contributions from a large and diverse developer base. The adaptation of the ISSM code base toward gradient computation by employing operator overloading through a user-defined type predates the existence of the tool discussed in this paper. It took place over an extended period of time during which additional features were added to ISSM's core capabilities, significant software technical improvements were made, and lastly the gradient computation specific adaptations were introduced, included complicated code refactoring currently far beyond the scope of the tool. Among the latter are:

- Templatization of certain container classes.
- Separation of passive and active floating point variables w.r.t. differentiation using alias definitions [2].
- Introduction of a thin MPI wrapper layer to refactor and encapsulate the logic for optional serial plain model, and serial or parallel gradient computation.
- The treatment of calls to (direct) linear solvers with special wrappers for the most efficient gradient computation.

6.1 Results

To give an idea of the potential impact of the tool, we compare its application to a version of ISSM that predates all these transformations (late 2011) with the version (mid 2014) of the code base when all major parts of the gradient computation were functional [17]. One could argue that a comparison crossing such a significant evolution of the code base is not terribly meaningful and that is certainly true for quantifying the output at a file level detail. However, it still gives a good idea for the *overall time savings* that could have been achieved had the tool been available four years ago to aid the adaptation. Table 2 shows the result of our analysis. We refer to a *fix* as a successfully applied source transformation. A *match* is a reported code occurrence which includes both the fixes and unchanged code. Thus for matches, the number of files reflect the usage context because of the specific file attribution given by Clang, e.g., for the use of macros.

Metric	Late 2011	Mid 2014
Files	912	857
LOC	66,573	80,044
Translation Units	298	254
#Explicit Conversion fix in #files	170 in 42	0
#Implicit Bool Conversion fix in #files	41 in 8	0
#Explicit Conversion match in #files	192 in 46	0
#Implicit Bool Conversion match in #files	44 in 10	0
#Conditional Assign match in #files	1,768 in 275	0
#If-Else Assign match in #files	98 in 9	0

Table 2: ISSM evolution w.r.t. problematic code constructs.

The drop in the number of C++ source files over time is mainly due to the templatization and other consolidations of the source code. These changes also reduced the number of non comment/blank LOC but that decrease is more than compensated by the addition of new features (e.g., implementing new approximations of the ice sheet physics), leading to growth in the LOC. Commensurate with the consolidation though is the drop in number of compile steps used here in a simple build time configuration, i.e. a configuration that leaves out certain portions of the total of 912 or 857 respectively C++ source and header files.

Most importantly, the remarkable amount of potential issues and actual transformations combined with their wide spread over distinct source files for the late 2011 version indicates the large amount of work that was necessary to manually locate and fix these problematic constructs. When one contrasts this result with the zero issues and transformation flagged in the mid 2014 versions, it becomes clear that almost all of the issues flagged represented legitimate problems that needed to be addressed in the 2011 code base in order to achieve functioning operator overloading for the ISSM code.

7 Conclusion and Future Work

Software maintenance is an ongoing process during the life cycle of a software package. In that context, adaptive maintenance can be a costly and time consuming task especially if major features need to be added. This makes non-trivial project-wide refactoring a necessity. Although semantic augmentation through operator overloading promises extensional features with minimal code changes, this type change may lead to compilation errors. In a previous paper, we identified problematic code constructs. Here, we identified further problematic constructs related to namespace resolution, and developed a Clang based tool that automatically identifies and fixes most of these issues. We described the design of this tool, taking advantage of Clangs matching capabilities and suggesting minimalistic code changes that ensure adherence of the resulting code to the C++ standard, also after a type transformation has been applied.

Applying this tool to the CFD software OpenFOAM, we were able to automatically remove several problematic code constructs that had been included in several hundred files thereby causing a great deal of error output. We also applied this tool to two versions of the ice flow modeling software ISSM, comparing a 2011 version of the code, with one that was manually refactored and prepared for automatic differentiation. In the 2011 version, we identified in total over 2000 occurrences of problematic code constructs. Had our tool been available then, the developers of ISSM would have been able to apply AD to their code base with significantly less effort, saving several month of coding drudgery. Consequently, this represents in our view an excellent argument for the utility of our tool.

In the future, we will explore solutions for ambiguous name resolution which are introduced by overloaded functions provided by user-defined types. In that context, we will also contact the OpenFOAM developers to discuss the reasoning behind the design of the `scalar` data type as well as potential alternative designs for compatibility with operator overloading. We will also work on exploring generic programming strategies, especially templates but also macro usage as we are still limited by missing alias information for the instantiation type in the Clang AST.

References

[1] Hervé Brönnimann, Guillaume Melquiond, Sylvain Pion, et al. The Boost Interval Arithmetic Library. In *Real Numbers and Computers*, pages 65–80, 2003.

[2] A. Mauer C. Bischof, A. Carle and P. Khademi. The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs. *IEEE Journal on Computational Science and Engineering*, 3:18–32, 1996.

[3] Gregory R Carmichael, Adrian Sandu, et al. Sensitivity Analysis For Atmospheric Chemistry Models Via Automatic Differentiation. *Atmospheric Environment*, 31(3):475–489, 1997.

[4] Clang. Clang-Tidy, 2015. http://clang.llvm.org/extra/clang-tidy/, last accessed: 04-2016.

[5] Al Danial. cloc, 2016. https://github.com/AlDanial/cloc/, last accessed: 04-2016.

[6] Michael Fagan, Laurent Hascoët, and Jean Utke. Data Representation Alternatives In Semantically Augmented Numerical Models. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, volume 6, pages 85–94, 2006.

[7] Makarand Gawade, K Ravikanth, and Sanjeev Aggarwal. Constantine: configurable static analysis tool in Eclipse. *Software: Practice and Experience*, 44(5):537–563, 2014.

[8] A. Griewank and A. Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics (SIAM), second edition, 2008.

[9] Alexander Hück, Christian Bischof, and Jean Utke. Checking C++ Codes for Compatibility with Operator Overloading. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, volume 15, pages 91–100, September 2015.

[10] International Standards Organisation: ISO/IEC 14764:2006. *Software Engineering - Software Life Cycle Processes - Maintenance*, 2006.

[11] International Standards Organisation: ISO/IEC 14882:2003. *Programming Languages - C++*, second edition, 10 2003.

[12] H. Jasak, A. Jemcov, and Z. Tukovic. OpenFOAM: A C++ Library for Complex Physics Simulations. In *International Workshop on Coupled Methods In Numerical Dynamics*, pages 1–20, 2007.

[13] Stephen C. Johnson. Lint, a C Program Checker. *Computer Science Technical Report*, 65, 1978.

[14] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E Nagel. Scout: A Source-to-Source Transformator for SIMD-Optimizations. In *Euro-Par 2011: Parallel Processing Workshops*, pages 137–145. Springer, 2012.

[15] Ajit Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ Programs through Demacrofication. In *28th IEEE International Conference on Software Maintenance (ICSM), 2012*, pages 98–107. IEEE, 2012.

[16] E. Larour, H. Seroussi, M. Morlighem, and E. Rignot. Continental scale, high order, high spatial resolution, ice sheet modeling using the Ice Sheet System Model (ISSM). *Journal of Geophysical Research: Earth Surface*, 117(F1), 2012.

[17] E. Larour, J. Utke, B. Csatho, A. Schenk, H. Seroussi, M. Morlighem, E. Rignot, N. Schlegel, and A. Khazendar. Inferred basal friction and surface mass balance of the Northeast Greenland Ice Stream using data assimilation of ICESat (Ice Cloud and land Elevation Satellite) surface altimetry and ISSM (Ice Sheet System Model). *The Cryosphere*, 8(6):2335–2351, 2014.

[18] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[19] LLVM. Clang Static Analyzer, 2015. http://clang-analyzer.llvm.org/, last accessed: 04-2016.

[20] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures. In *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), 2011*, pages 300–307. IEEE, 2011.

[21] Nicholas Jie Meng, Diane Kelly, and Thomas R Dean. Towards The Profiling Of Scientific Software For Accuracy. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 257–271. IBM Corp., 2011.

[22] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a Component Source Transformation System for Differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, 2010.

[23] Roger P Pawlowski, Eric T Phipps, and Andrew G Salinger. Automating Embedded Analysis Capabilities And Managing Software Complexity In Multiphysics Simulation, Part I: Template-Based Generic Programming. *Scientific Programming*, 20(2):197–219, 2012.

[24] Dan Quinlan. ROSE: Compiler Support For Object-Oriented Frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[25] N. Safiran and U. Naumann. Toward Adjoint OpenFOAM. Technical report, Citeseer, 2011.

[26] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, fourth edition, 2013.

[27] M. Towara and U. Naumann. A Discrete Adjoint Model for OpenFOAM. *Procedia Computer Science*, 18(0):429 – 438, 2013. 2013 International Conference on Computational Science.

[28] Eelco Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831 – 873, 2005. Reduction Strategies in Rewriting and Programming special issue.

[29] Hyrum K Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. Large-Scale Automated Refactoring Using ClangMR. In *29th IEEE International Conference on Software Maintenance (ICSM), 2013*, pages 548–551. IEEE, 2013.

[30] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. ARC++: Effective Typestate and Lifetime Dependency Analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 116–126. ACM, 2014.

[31] Beckett Yx Zhou, Tim A. Albring, Nicolas R. Gauger, Thomas D. Economon, Francisco Palacios, and Juan J. Alonso. A Discrete Adjoint Framework for Unsteady Aerodynamic and Aeroacoustic Optimization. In *AIAA Aviation*. American Institute of Aeronautics and Astronautics, 2015.