COLLEGE OF COMPUTING
GEORGIA INSTITUTE OF TECHNOLOGY

Comparison of Metaprogramming and Template Programming Solutions for Invariant Maintenance through Implicit Invocation

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
spencer@cc.gatech.edu

Jonathan Gdalevich
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
lordebon@cc.gatech.edu

Abstract

Large software systems commonly contain multiple interdependent components. When independent components change, dependent components must change as well in order to establish system invariants. This scheme leads to a variety of approaches for components to communicate with each other to maintain the invariants. One promising way to do so is to automatically generate implicit invocation code for maintaining the invariant between dependent and independent components. However, since a complex system could have many invariants and performance requirements, the generated code must have a small runtime overhead. This paper explores eight separate approaches for the implementation of implicit invocation invariant maintenance in C++ using compile-time metaprogramming via OpenC++ and generative programming with C++ templates.

1 Introduction

1.1 Motives

An invariant is a relationship between components in which the state of one component, the dependent, is based on the states of other independent components. For instance, if a variable a in component A must always be twice the value of variable b in component B, A is a dependent component that depends on independent component B. The formula a = 2 * b is therefore the invariant. The easiest way to specify the relationship is with explicit invocation in which A and B know about each other in order to communicate the event of change in B. However, explicit invocation results in high coupling between system components complicating maintenance and addition of features. Another method of invariant maintenance is implicit invocation [1] where instead of invoking a direct procedure, the independent component announces the event to any

dependent components responsible for maintaining the invariant. In implicit invocation, B would announce that it has changed to all system dependent components. A would detect the change and retrieve the value of B in order to change itself. The benefit of implicit invocation can be seen with an addition of component C dependent on B. With explicit invocation, B would have to keep track to both A and C so that both could be notified in case B changes. Conversely, implicit invocation allows B to have no knowledge of A or C but to simply announce changes so that both A and C detect the change and update themselves as necessary.

While beneficial, the use of implicit invocation for invariant maintenance raises two questions. First, there are many approaches to implementing implicit invocation. Some require extra objects to hold the invariant rules and detect changes while others distribute invariant maintenance between multiple dependent and independent components. Likewise, while certain approaches allow invariant maintenance when dependent and independent variables are switched, others only support static invariance. Second, a complex system can support hundreds or thousands of invariants making it difficult to keep track of dependent and independent components. Therefore, generative programming techniques must be employed to implement implicit invocation from rules written in a declarative language such as OCL [3] or another constraint language. Two appealing C++ implementation techniques are metaprogramming and template programming. When used with different implicit invocation approaches, the advantages and disadvantages of each technique must be analyzed and compared to determine the best one for each particular situation.

One way to implement implicit invocation within invariant maintenance is through the use of metaprogramming as described in [4]. Metaprogramming involves the manipulation of program code, data, or objects either at compile time or runtime by another program. In the a = 2 * b example, metaprogramming can be used to generate the code required for B to broadcast its change event and for A to receive the event and update itself. Moreover, the same metaprogram can generate update code for both A and C if given the description of the invariant, freeing the developer from having to keep track of dependent and independent components of each invariant. Furthermore, the metaprogram can be written to generate different implementations based on the nonfunctional requirements such as reuse or performance. For instance, a change in B would change A to 2B while a change in A not caused by a change in B would not effect B's value. On the other hand, a change in B would change C to 3B but a change in C, not caused by a change in B, would also change B through the invariant B = 1/3C. Given the invariant rules and the components, it is perfectly valid to expect the metaprogram to implement all three without errors within the resulting program. However, metaprogramming, aside from template programming, is not built-into C++ and usually requires a separate tool or compiler for executing the metaprogram.

Contrasted to metaprogramming, template programming has been an integral part of ANSI/ISO C++ since early versions [5]. The main purpose of templates is to allow one component or structure to take in or use different datatypes determined at compile time. This permits templates to be used in generating an implementation of implicit invocation where the dependent component does not know the invariant's independent components until compile time. Without templates, the developer would be forced to specify the exact type of the independent component in the code. When combined with generative programming, templates that support implicit

invocation can be generated based on invariant constraints before compiling the entire program. Best of all, since most C++ compilers support template programming, no extra tools or compilers are required for compiling the generated code with the system components.

The purpose of this research is to determine which programming technique under which approach would be most beneficial in implementing implicit invocation for invariant maintenance. The benefits can be divided into two broad categories of quantitative and qualitative analysis. Quantitative analysis is measured through compile and runtime comparison as well as the analysis of final assembly code. Qualitative analysis, on the other hand, is harder to quantify and will be determined by the amount of code the programmer has to modify and the lines of new code written to take advantage of each technique and approach. Combined, quantitative and qualitative analysis will be used to determine the best approach to implementing implicit invocation for invariant maintenance.

1.2 Related Work

Besides implicit invocation, there are a number of techniques to avoid direct references between dependent and independent components. Researchers at the University of Twente in the Netherlands, developed objects known as Abstract Communication Types (ACTs) that contains abstract senders and receivers for handling communication between system components [6]. ACTs also provide mechanisms for synchronization of messages and for reflection upon a message. They are implemented in the programming language Sina, which is not as widespread or popular as C++. On the other hand, Robert DeLine's Flexible Packaging approach [7] allows the user to determine the exact nature of the interactions between components at integration time by separating each component into functional and interactional parts. Due to a large runtime overhead from message channeling mechanism built into each component, Flexible Packaging results in a significant runtime cost of up-to 8% and should be avoided for invariant maintenance. Finally, Kevin Sullivan and David Notkin from the University of Washington describe a case study in the implementation of *mediators* to support invariant maintenance between components [8], [9]. A mediator is a separate component that contains implicit or explicit references to all components in the invariant as well as the relationship between dependent and independent components. The components themselves know only about the mediator and announce any changes to it. Inside the mediator, changes from independent components are received and passed on to dependent components. Any change to an invariant is implemented within the mediator and does not require changing components aside from adding or removing mediator references. All the mediator needs is to be informed of the change in a component and given an ability to retrieve the new value.

Metaprogramming is divided into runtime metaprogramming, also known as *reflection*, and compile time metaprogramming. Reflection is the more interesting since it allows interaction and modification of objects, components, and structures that do not exist as compile time. Languages like Smalltalk and Lisp facilitate reflection through the use of an interpreter implemented in the same language. Brian Foote and Ralph Johnson demonstrate the ease and usefulness of reflection within Smalltalk-80 [10] through the construction of monitors, distributed objects, and futures, and experimentation with new inheritance, delegation, and protection schemes. Moreover, Fred Rivard uses Smalltalk reflection to implement an invariant

constraint system resulting in a 2% compile time increase in exchange for a 9% reduction at runtime [11].

Unlike Smalltalk and Lisp, Java and C++ do not feature extensive built-in reflection facilities. While there is a Java reflection API [12], it is limited to determining the signature of objects and methods, but not the manipulation of those components. Nevertheless, OpenJava, from the University of Tsukuba, is a class-based macro system that allows metaprogramming by manipulating an abstract syntax tree containing the logical structure of a Java program [13]. Likewise, a variety of tools and techniques have been developed to provide metaprogramming in C++. For example, [14], [15] and [16] provide directions and libraries for implementing different types of metaprogramming for C++. However, one of the most popular and useful metaprogramming tools for C++ is Shigeru Chiba's OpenC++ metaobject protocol [17]. Based on the Gregor Kiczales' suggestion for implementing abstraction via metaprogramming [18], OpenC++ provides an abstract syntax tree to represent C++ code and an API to manipulate it. Although, it does not use true runtime reflection, OpenC++ allows the manipulation of source code without any direct runtime cost. In fact, Michigan State University's TRAP/C++ uses OpenC++ to implement facilities for selecting and incorporating new behavior at run time into C++ systems [19].

An alternative to metaprogramming for the implementation of implicit invocation in invariant maintenance is template programming. Also known as "programming at compile time", templates allow the compiler to determine certain variable and object types at compile time. This can be used to divide architecture into layers containing different parts of an invariant. For example, GenVoca generators construct applications from reusable layers through the use of C++ templates [20]. The generators are used to implement structures called mixin-layers that combine commonly used parts of objects into a single structure for faster access without subtyping [21]. Mixin-layers are in-turn used within the DYNAMO project where the order of the layers is specified to support invariant maintenance through implicit invocation [22], [23], [24]. To do this, C++ template classes, containing a template parameter that is the class's supperclass, allow dependent components to access features of independent components without resorting to the run-time cost of using a pointer.

1.3 Research Questions

This study attempts to answer the following questions about implicit invocation implementation of invariant maintenance. Which technique, metaprogramming or template programming, provides the better solution when applied to an invariant maintenance system as evaluated by quantitative and qualitative criteria including performance, ease-of-use, and simplicity of input?

1.4 Paper Map

The next section describes the approach used for the case study. It includes evaluation criteria and an account of why this research is unique. Section 3 contains the case study with the description of each approach. Section 4 contains the results of the case study including the analysis of generated assembly code. Section 5 discusses the results including their implications and suggestions for future work. Finally, section 6 concludes the paper with a reflection on the research study.

2 Approach

2.1 Evaluation Criteria

To evaluate and compare metaprogramming with template programming for the implementation of implicit invocation of invariant maintenance, solutions to the same problem were implemented using different techniques and approaches. Afterwards, each solution was analyzed based on predefined quantitative and qualitative criteria. Quantitative criteria included time measurements of runtime and compile time, count of generated lines of assembly code, count of lines of C++ code required to implement the method in the main() method, and the size of the generated executable in bytes. Qualitative criteria is comprised of personal evaluations of metaprogramming and template programming. Those include how easy each technique was to install and use and how well each technique fits the programmer's mindset about the problem. While subjective, qualitative conclusions are based on real life experience described in the **Case Study**.

2.2 Metaprogramming

Unlike Lisp or Smalltalk, C++ does not support a built-in metaprogramming system aside from template metaprogramming. Therefore, OpenC++, a third party technology, was used to implement metaprogramming solutions to the invariant maintenance problem. OpenC++ is an independent C++ compiler extension that allows, among other features, the detection of assignment within C++ code in order to generate and insert new code for the implementation of invariant maintenance. First, it translates an input C++ program into an abstract syntax tree containing all program structures, objects, and variables. Afterwards, the tree is manipulated based on a C++ metaprogram written using the OpenC++ API. Using its own compiler, OpenC++ transforms the original C++ program into a new C++ program that includes user manipulations described in the OpenC++ metafile.

For invariant maintenance, all the approaches studied use OpenC++ to detect a change in the independent variable and to call a function that changes the corresponding dependent variables. Furthermore, implicit invocation between the components is generated by OpenC++ based on the specific metaprogram input to the compiler. The only parts requiring user assistance are the code in the main() method that creates instances of classes and connects them if necessary and the declaration of a class required for OpenC++ manipulation represented by **metaclass <metaclass name> <class name>;** line of code.

As shown by **Figure 1**, OpenC++ takes the unmodified C++ base-level program and a special C++ metaprogram as input and outputs the executable. First, the OpenC++ compiler uses the C++ preprocessor to transform the metaprogram into a series of libtool object and meta-level object files. Afterwards, the object files are used to modify the C++ base-level program into a new program that includes generated code. This program, stored within .ii files, is then used to generate the executable.

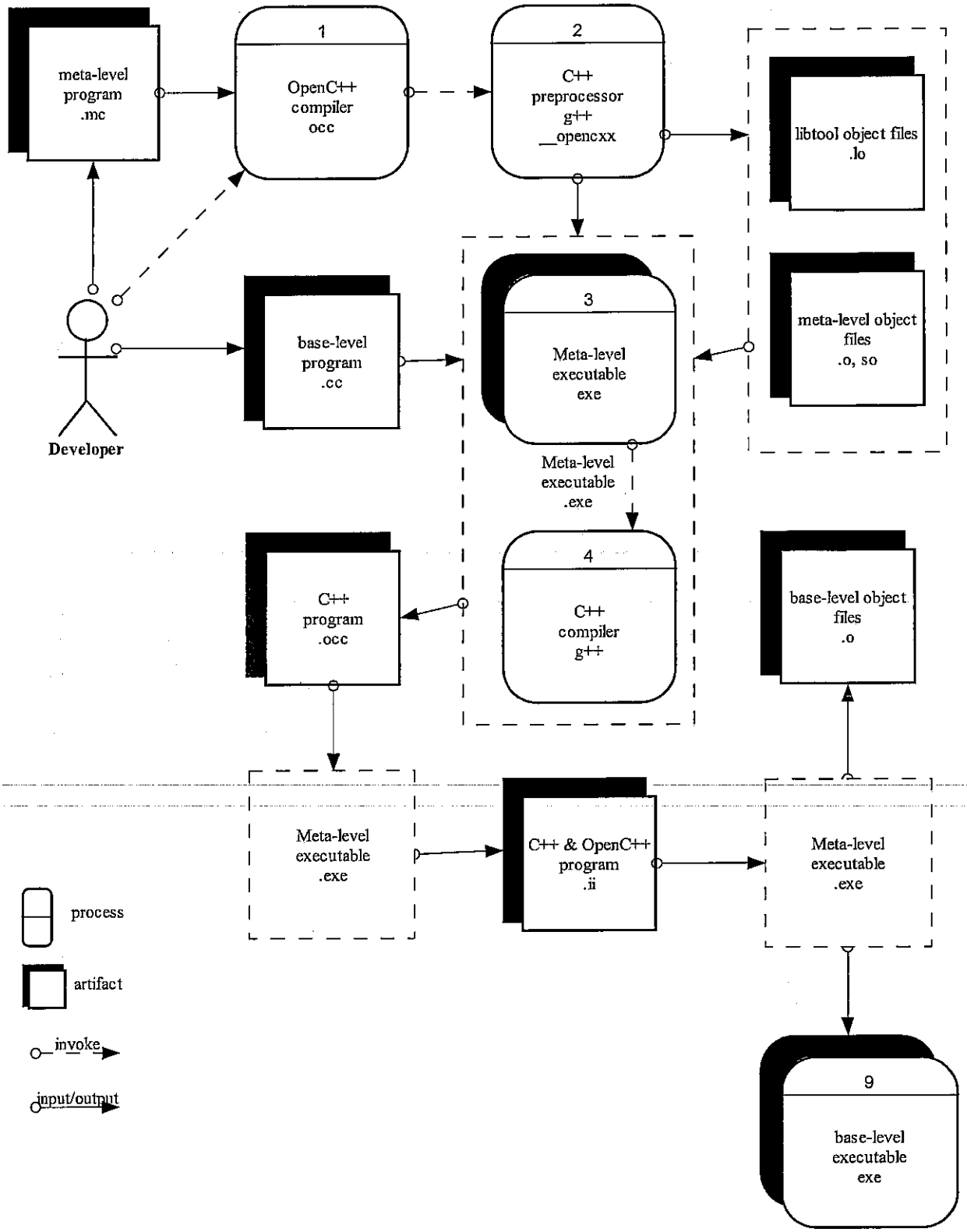

Figure 1 OpenC++ Development Process

OpenC++ was selected due to its popularity within academia and its available support through a network of users. Both Georgia Institute of Technology and Michigan State University apply

OpenC++ in teaching languages and research. Likewise, an active newsgroup devoted to OpenC++ is maintained by SourceForge.net with over eighty users. According to SourceForge.net, OpenC++ is a mature project with two administrators, 27 developers, and over 700 CVS commits. Further description and evaluation of advantages and disadvantages of OpenC++ can be found in **Appendix A**.

2.3 Template Programming

Contrasting with metaprogramming, templates are an integral part of C++ requiring no separate tools or compilers. Since they are Turing-complete, templates can be used to implement any approach implemented with metaprogramming. For this study, templates were used to wrap components in order to provide interfaces for component communication without explicit invocation. Specifically, templates allowed the compiler to bind communicating components at compile time instead of at runtime. Time constraints allowed for only one implementation based on mixin-layers from the DYNAMO project. Please see the **Case Study** for better description of template programming implementation and **Appendix B** for advantages and disadvantages of template programming.

3 Case Study

3.1 Introduction

To compare metaprogramming to template programming, nine different solutions were implemented. The problem each solution is trying to solve is the invariant of temperature conversion between Celsius and Fahrenheit. The invariant formula for converting from Celsius to Fahrenheit is **Fahrenheit = (1.8 * Celsius) + 32**. In OCL, this would be represented as:

context F inv:

self.f = (C.c * 1.8) + 32

from within the Fahrenheit class called F referring to the Celsius class called C. On the other hand, converting from Fahrenheit to Celsius is accomplished using the formula **Celsius = (Fahrenheit – 32) / 1.8**. In OCL, this would be represented as:

context C inv:

self.c = (F.f - 32) / 1.8

from within the Celsius class called C and referring to the Fahrenheit class called F. While some approaches easily support conversion in from Celsius to Fahrenheit and from Fahrenheit to Celsius, others are best at supporting conversion in only one direction. All solutions are implemented using GNU C++ since it has a complete template programming support, and it provides for a number of metaprogramming tools.

The nine approaches studies are divided into eight metaprogramming and one template programming solution. Metaprogramming includes one-way and two-way solutions for status variables and pointers. It also includes solutions involving inheritance, mediators, and distribution. The single template programming solution is based on status variables in one direction. Source code for all solutions can be found at http://www.cc.gatech.edu/dynamo/tools/c2f.zip.

3.2 Metaprogramming

3.2.1 Status Variable (One-way)

The one-way status variable approach closely models DYNAMO's template design. The differences are due to the lack of wrapper classes in the metaprogramming solution. Instead, C and F act like C_Top and F_Bot respectively. Instead of F_Bot being given a template to C_Top, this approach provides F with the address of the instance to C and binds the instance of F to C as an Updater. Implicit invocation is used to ensure that C does not know who is bound to it but simply expects an Updater.

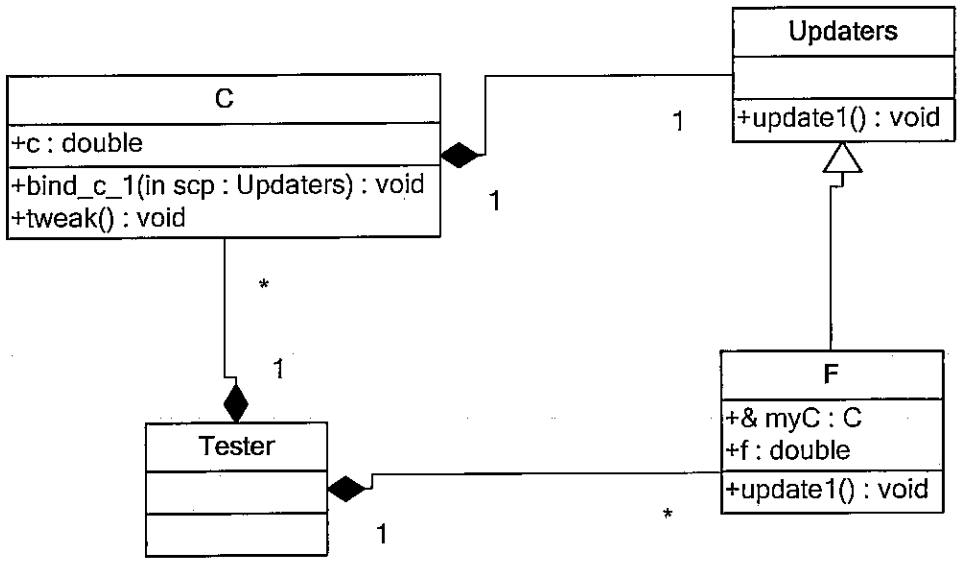

Figure 2 One-way status variable class diagram

The method tweak() calls the update1() method that proceeds to retrieve the value of c and then change the value of f. This method is generated by metaprogramming and called every time the value of C changes. The main() method requires the presence of the following code:

```
C myC;
F myF(myC);
myC.bind_c_1 (&myF);
```

Note that neither temperature requires initialization at creation. This will cause C to remain undefined if the value F changes before C is initialized.

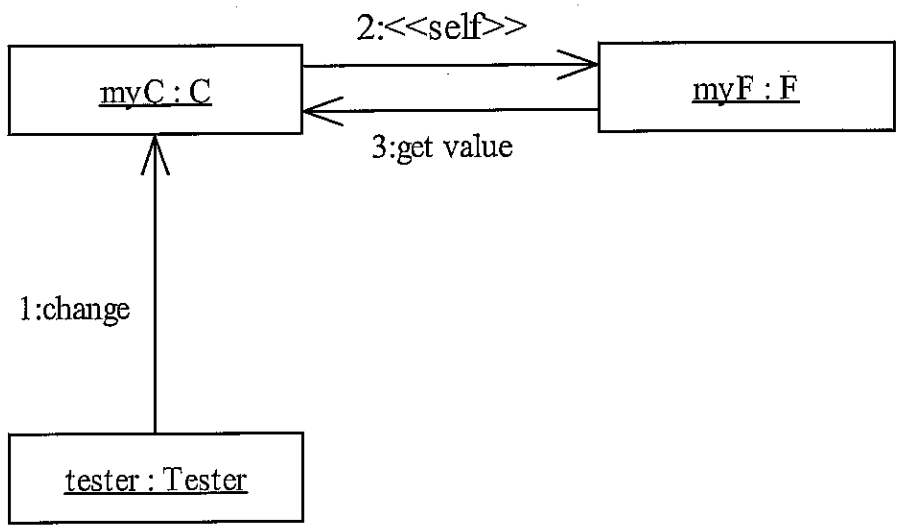

Figure 3 One-way status variable collaboration diagram

When the value of c changes, tweak() inside of C is called. It gives a pointer to C to F which uses the update1() method to retrieve the value of c from C and update f.

3.2.2 Status Variable (Two-way)

Two-way status variable solution only differs from one-way solution in that both C and F inherit the update() method from the Updater class and employ it in binding to each other. The binding results in a perfect implementation of implicit invocation where neither C not F directly reference one another except through a third party represented by the Updater.

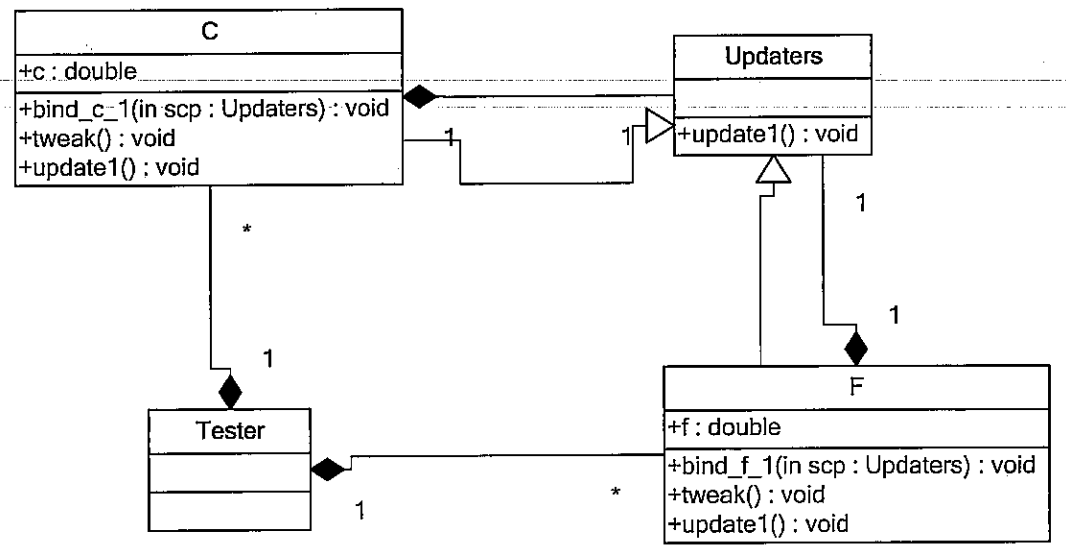

Figure 4 Two-way status variable class diagram

Moreover, the additional code within the main() method is limited to two new lines that bind objects together.

C myC;
F myF;

myC.bind_c_1 (&myF);
myF.bind_f_1 (&myC);
Note that the original C and F class declarations are kept intact.

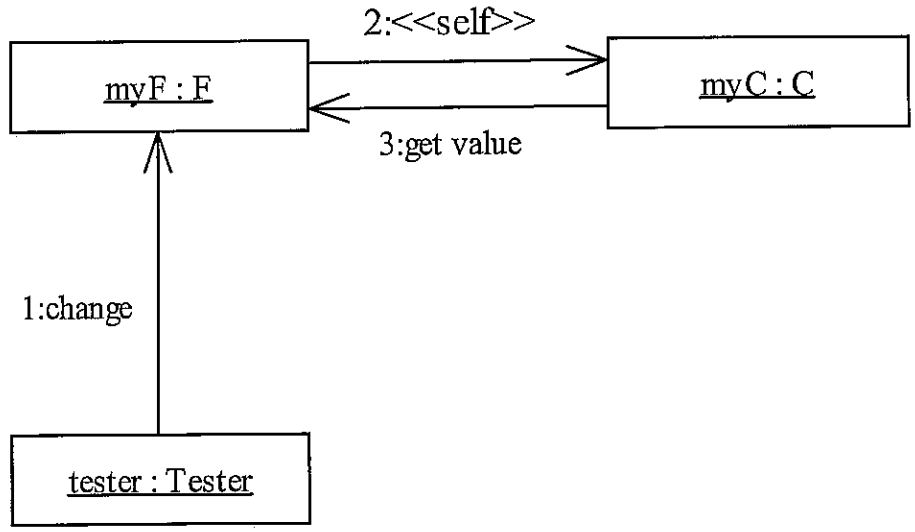

Figure 5 Two-way status variable collaboration diagram

The interaction between objects is same is in Status Variable one-way except C and F are reversed if f is the independent variable.

3.2.3 Pointer (One-way)

The pointer solutions differ from status variables in that they do not implement implicit invocation through the use of inheritance. Instead, C contains a pointer to F through which it knows about the F update method. This method is called when a change in c occurs and is given a reference to C. Through this reference, F is able to access the new value of c and use it to change the value of f. This illustrating an implementation of indirect invocation since the pointer is indirect and can easily be rebounded.

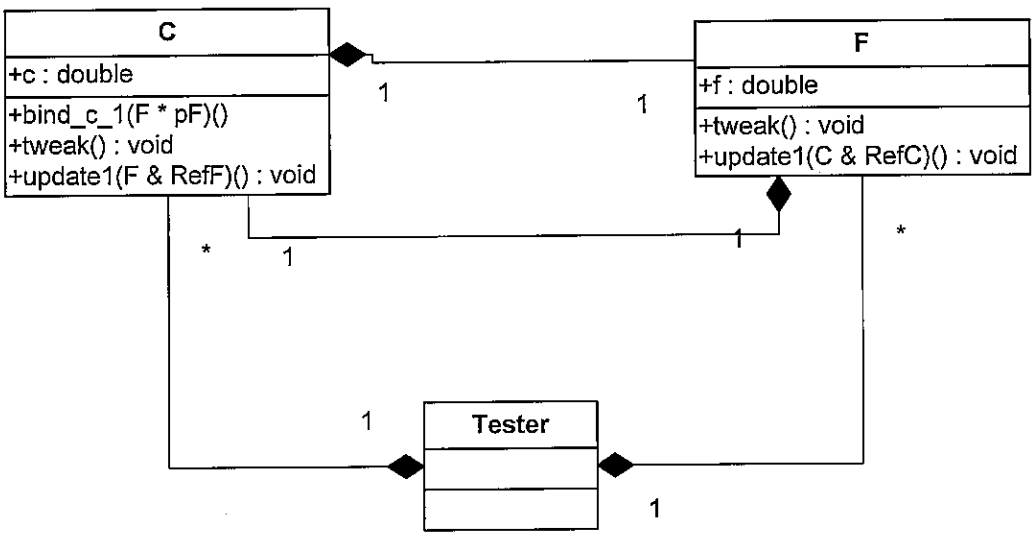

Figure 6 One-way pointer class diagram

Unlike one-way status variable, the main() method does not require that an instance of C be passed into F.

C myC;
F myF;
myC.bind_c_1 (&myF);

Note that only one line of code needs to be added to the original declarations of C and F.

The sequence of message for one-way pointer is exactly the same as the sequence of messages for one-way status variable.

3.2.4 Pointer (Two-way)

Just like one-way pointer is similar to one-way status variable, two-way pointer is similar to two-way status variable. The only difference is the use of indirect instead of implicit invocation for communication between C and F. Instead of having each object contain a reference to a third party they each inherit from, the objects simply hold pointers to one another.

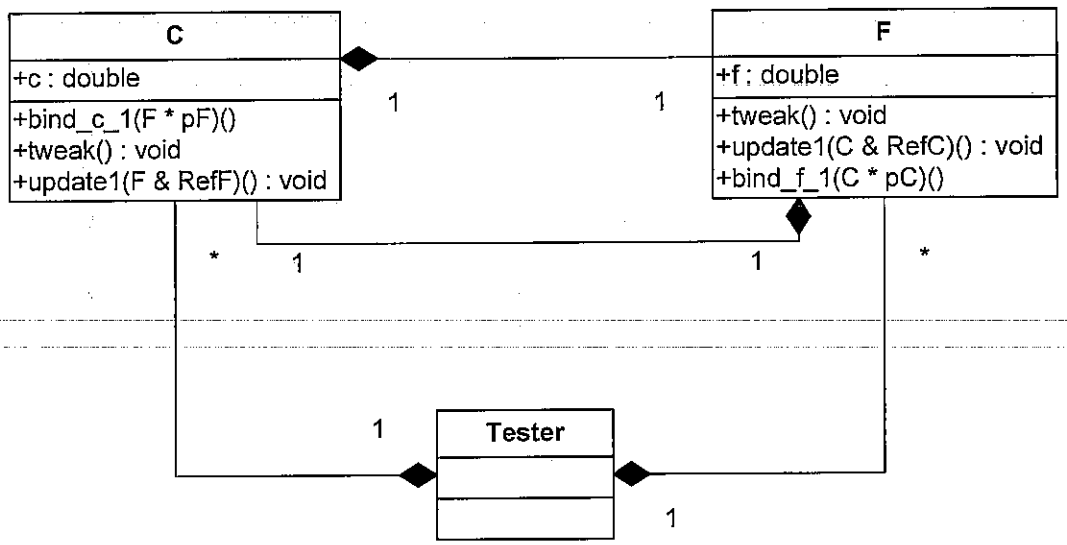

Figure 7 Two-way pointer class diagram

The resulting main() method adds an extra binding for backward invariant.

C myC;
F myF;
myC.bind_c_1 (&myF);
myF.bind_f_1 (&myC);

Note that this solution is exactly the same as two-way status variable and requires the implementation of two extra lines of code.

The sequence of message for two-way pointer is exactly the same as the sequence of messages for two-way status variable.

3.2.5 Inheritance (One-way)

The inheritance approach is designed to optimize one-way Status Variable approach through improvements within the generated assembly code. The only difference from Status Variable is the inheritance of C by F.

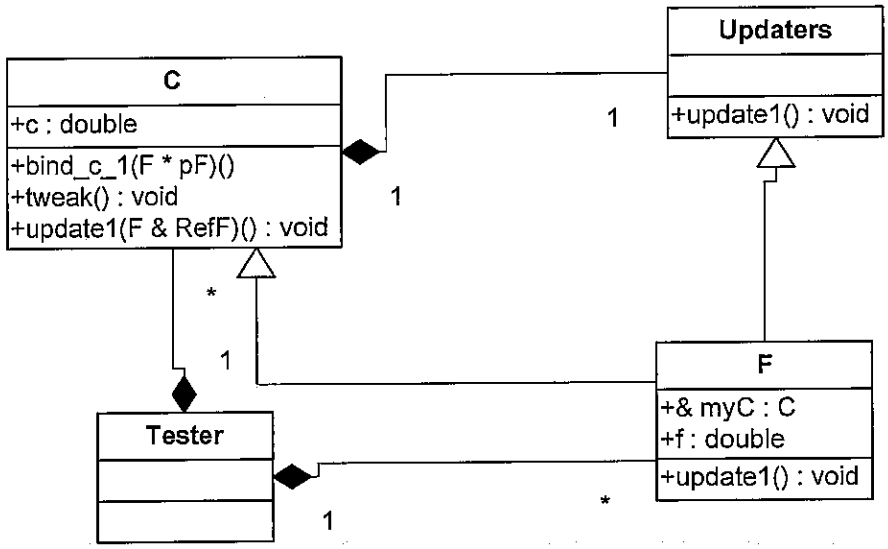

Figure 8 One-way inheritance class diagram

This produced a reduction in generated assembly code by 141 lines but no noticeable difference at runtime or compile time. Please see the Results section for further discussion of the assembly code.

The main() method remained exactly the same as in one-way status variable.

```
C myC;
F myF(myC);
myC.bind_c_1 (&myF);
```

The sequence of messages for inheritance is exactly the same as the sequence of messages for one-way status variable. However, the delegating component F is now a subclass of the delegate component C.

3.2.6 Mediator (Two-way)

Instead of encapsulating invocation within existing classes, the mediator approach moves all communication, along with invariant maintenance code, into a brand new object. Known as a mediator, this object encapsulates references to C and F in addition to the invariant formula. Moreover, the mediator approach allows for a simple two-way invariant by containing two functions with the corresponding invariant formulas. Finally, the resulting C and F classes contain the tweak() method that allows the Mediator to be informed of a change in the invariant.

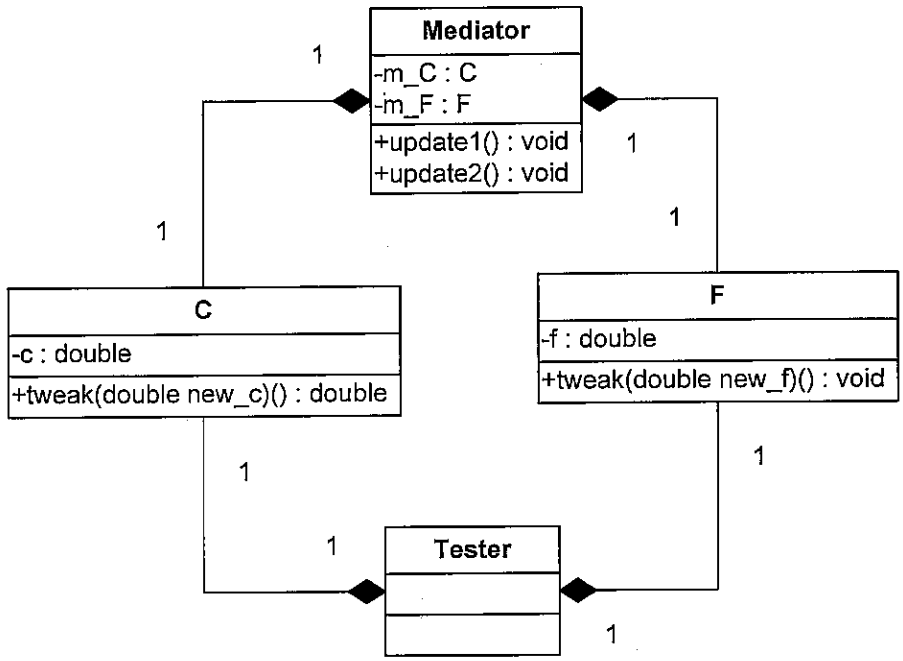

Figure 9 Two-way mediator class diagram

The main() method adds three lines of code in which as instance of the mediator class is created and given references to C and F objects while instances of the components are given instances to the mediator for upward communication.

```
C myC;
F myF;
Mediator myMediator(myC, myF);
myC.bind_Mediator(&myMediator);
myF.bind_Mediator(&myMediator);
```

Note that the original C and F class declarations are kept intact.

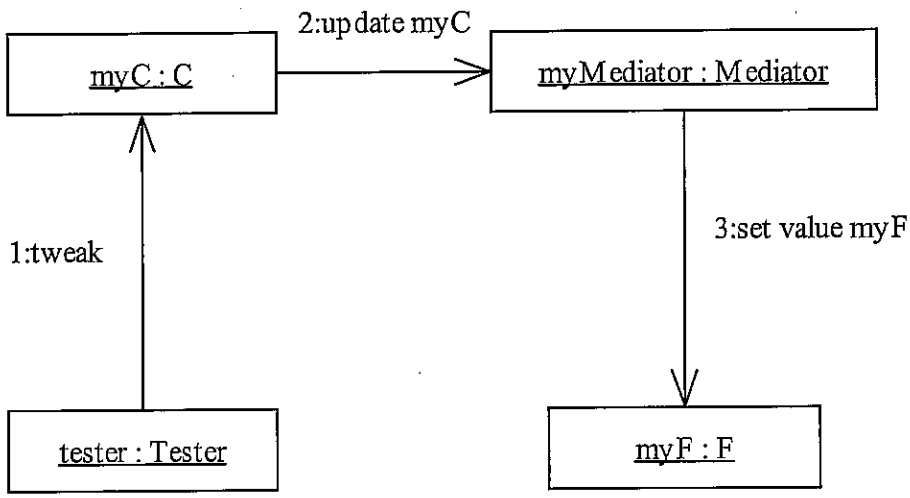

Figure 10 Two-way mediator collaboration diagram

In the mediator solution, when either c or f changes, the component alerts the Mediator through the tweak() method and the Mediator updates the other component with the new value.

3.2.7 Mediator (Optimized) (Two-way)

The Mediator Optimized solution is an optimization of the original mediator. In it, the Tester alerts the Mediator of the change in the invariant without going through one of the invariant components. The resulting C and F classes are completely unchanged with the exception of a single line of OpenC++ code required to run the tool like **metaclass MetaClass C;**.

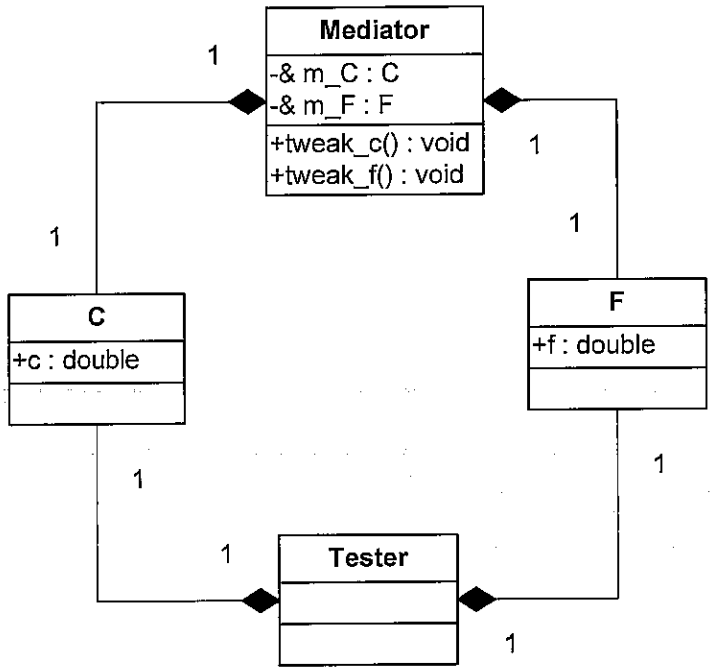

Figure 11 Two-way mediator optimized class diagram

The main() method adds one line of code in which as instance of the mediator class is created and given references to C and F objects.

C myC;
F myF;
Mediator myMediator(myC, myF);
Note that the original C and F class declarations are kept intact.

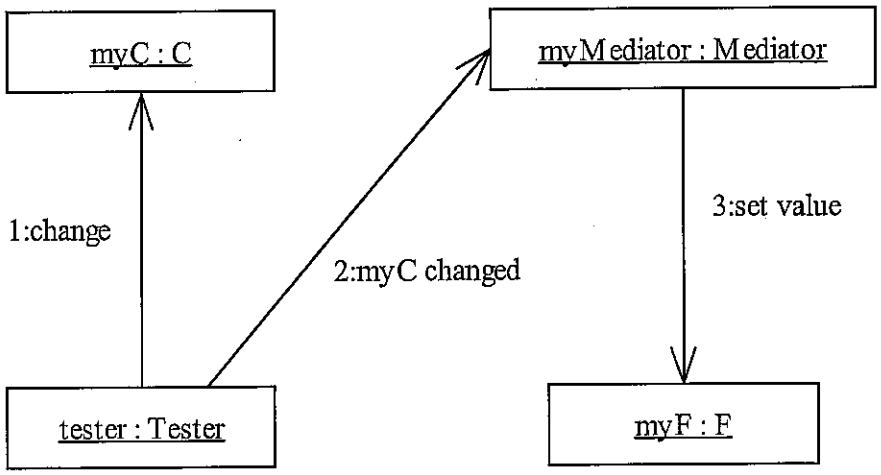

Figure 12 Two-way mediator optimized collaboration diagram

In the mediator optimized solution, when either c or f changes, the Mediator is alerted of the change and updates the other component with the new value.

3.2.8 Distribution (One-way)

Unlike other approaches that place invariant maintenance within a single object, distribution calls for a solution in which the formula or formulas for invariant maintenance are distributed among multiple components. The temperature conversion problem is too simple to implement any meaningful distribution. Nevertheless, an approach in which invariant maintenance is encapsulated within the same object as implicit invocation would closely model real distribution. Based on one-way pointer approach, distribution lets C maintain a pointer to F. However, instead of using the pointer to invoke F when a change occurs, C calculates the new value and updates f. This way, F is kept unaware of C and clean from any extra code. In fact, distribution results in the smallest generated assembly code and executable file of the nine approaches.

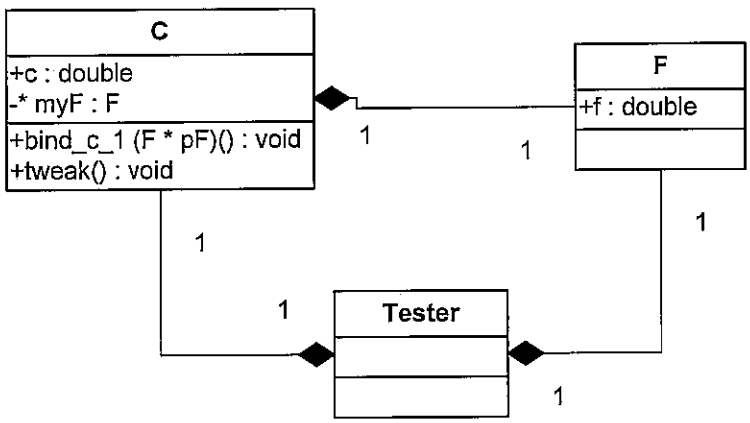

Figure 13 One-way distribution class diagram

The main() method requires the addition of only one new line of code that binds the reference to F to an instance of C.

C myC;
F myF;

myC.bind_c_1 (&myF);
Note that the original C and F class declarations are kept intact.

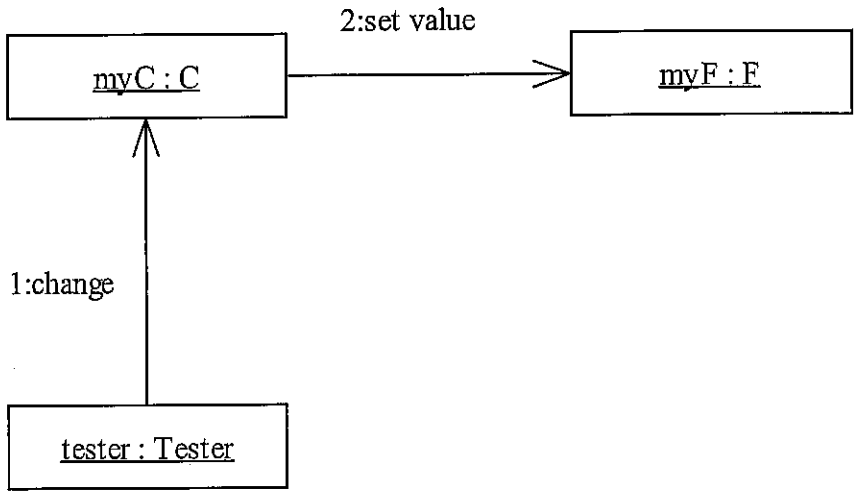

Figure 14 One-way distribution collaboration diagram

When c changes, C calculates and updates f with its new value.

3.3 Template Programming

3.3.1 Template Status Variable (One-way)

The template architecture is based on layers containing the dependent and independent components. It is described in depth by "Metaprogramming Compilation of Invariant Maintenance Wrappers from OCL Constraints" from the DYNAMO project [23]. For the solution of the Calculus to Fahrenheit problem, only the invariant maintenance formula from the paper's sample was modified.

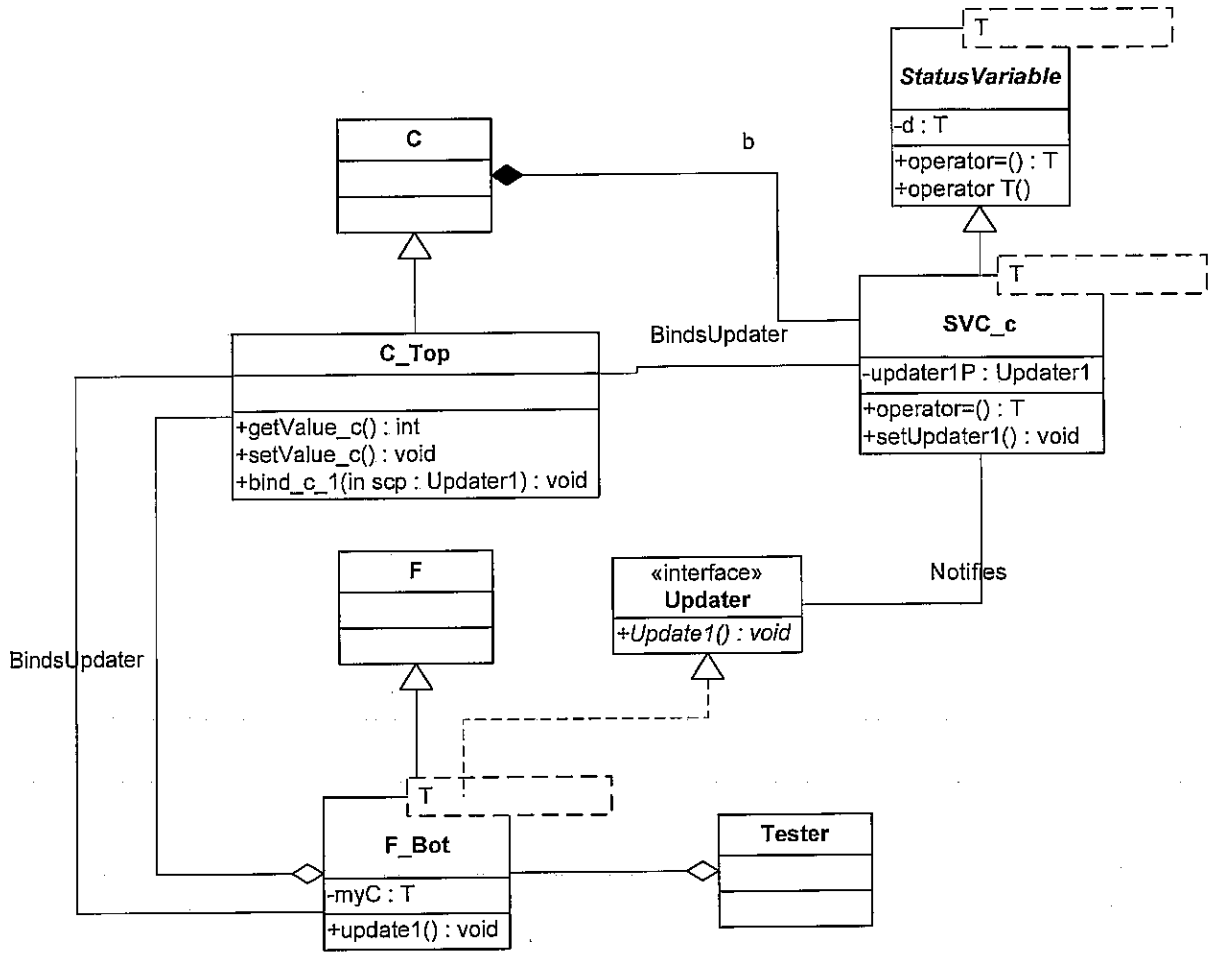

Figure 15 One-way template status variable class diagram

While this solution includes the most classes and methods, templates ensure that no runtime penalty is incurred. The main method requires the presence of the following initialization and binding code:

F_Bot<C_Top> myF(32);
C_Top* lowerP = myF.getLowerP();

Note the initialization of the Fahrenheit component with the temperature before the creation of the Celsius component.

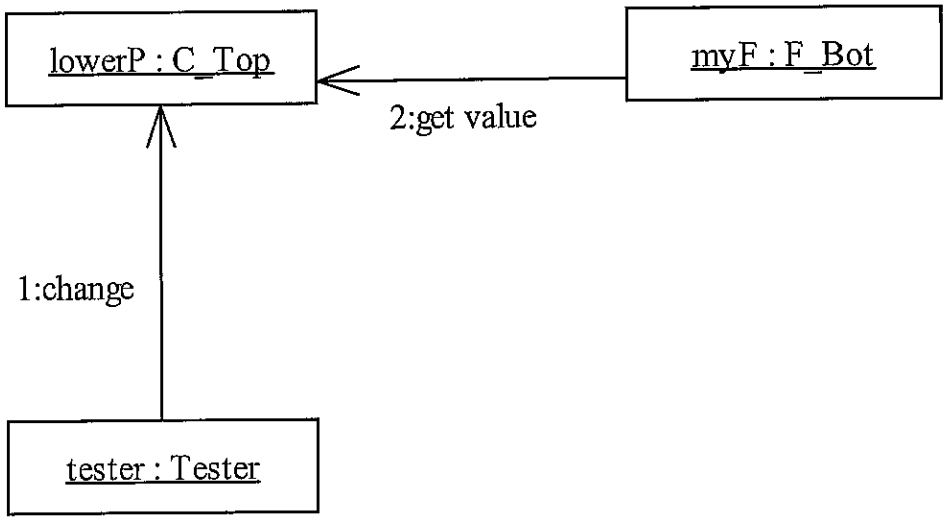

Figure 16 One-way template status variable collaboration diagram

Although the template programming solution looks much more complex than the one-way status variable solution from metaprogramming, the latter is in fact modeled on the former. When c changes, the assignment is overwritten to notify F_Bot. F_Bot in turn, retrieves the value of c from C_Top and uses it to change the value of f.

3.4 Case Study Uniqueness

While both metaprogramming and template programming parts of the case study are based on previous research described in the **Related Works** section, their comparison adds uniqueness to this research. Likewise, the study is fairly broad, containing multiple approaches to invariant implementation including those mentioned, but not implemented, in [23]. This allows for a more definitive conclusion that takes into account nine different approaches using the two different techniques. Finally, the focus on qualitative analysis in addition to quantitative analysis results in a broader evaluation. Usually, the focus of research is on the technical aspects resulting in a technologically interesting but not very useful conclusion if the technology or technique has commercial uses. However, focusing on ease-of-use allows future researches to select an approach that can quickly be translated into consumer-friendly applications.

3.5 Summary

The nine solutions presented can be divided into several categories. The broadest one is the technique used for implementation like metaprogramming or template programming. Next, the solutions can be separated into one-way or two-way invariants, ease of implementation, and whether or not extra objects were used in the solution.

4 Results

The following quantitative results were obtained by implementing each of the case study solutions on a single system. The system used is an AMD Athlon64 3200 processor with 1 Gigabyte of RAM. The solutions were compiled and run in the Cygwin 1.5.18-1 environment, under Windows 2000 SP4. GNU gcc 3.4 was used for all compilations.

4.1 Compile and Runtime

Amount of time to compile and run were measured using the "time" command within the Cygwin environment. For runtime, the average of the second, third, and forth runs was taken to avoid page miss error during the first run. The optimization flag -O3 was used during all compilations except when the option –S was used to generate assembly code. All compile code is based on the main() method having a loop that runs 10,000 times changing the value of the independent variable twice each time through the loop.

Figure 17 contains all data from the case study including time measurements, executable sizes, and line counts for assembly code. While the template programming solution has the shortest compile time, its executable is the largest and it generates the most assembly code. For metaprogramming, two-way solution have larger executables and more assembly code except for mediators. Inheritance is the largest of the metaprogramming one-way solutions while distribution is the smallest in the size of the executable and the length of assembly code. Nevertheless, runtime for all solutions is almost exactly the same when measured over multiple executions. Conceivably, the invariant tested is too simple and the test bed too powerful to obtain a valid difference.

Architecture	Status Variable	Status Variable	Pointer	Pointer	Inheritance	Mediator	Mediator (Opt)	Distribution	Status Variable
Direction	One Way	Two Way	One Way	Two Way	One Way	Two Way	Two Way	One Way	One Way
Method	metaprogramming	metaprogramming	metaprogramming	metaprogramming	metaprogramming	metaprogramming	metaprogramming	metaprogramming	template
Compile Time	real 0m16.186s	real 0m17.646s	real 0m19.250s	real 0m16.862s	real 0m16.268s	real 0m17.708s	real 0m27.088s	real 0m15.863s	real 0m0.985s
	user 0m15.727s	user 0m16.699s	user 0m16.038s	user 0m16.096s	user 0m15.699s	user 0m16.184s	user 0m16.099s	user 0m15.972s	user 0m0.923s
	sys 0m5.604s	sys 0m5.817s	sys 0m5.766s	sys 0m5.555s	sys 0m6.201s	sys 0m5.495s	sys 0m4.994s	sys 0m5.351s	sys 0m0.288s
Run Time	real 0m0.025s	real 0m0.026s	real 0m0.026s	real 0m0.025s	real 0m0.025s	real 0m0.030s	real 0m0.025s	real 0m0.025s	real 0m0.025s
	user 0m0.030s	user 0m0.035s	user 0m0.030s	user 0m0.046s	user 0m0.030s	user 0m0.030s	user 0m0.030s	user 0m0.030s	user 0m0.030s
	sys 0m0.030s	sys 0m0.015s	sys 0m0.030s	sys 0m0.015s	sys 0m0.030s	sys 0m0.030s	sys 0m0.030s	sys 0m0.015s	sys 0m0.030s
Size of Executable	478,237 bytes	478,939 bytes	477,571 bytes	477,880 bytes	478,454 bytes	477,999 bytes	477,467 bytes	476,824 bytes	479,628 bytes
Lines of Assembly	384	512	289	378	418	437	304	269	559

Figure 17 Quantitative results of case study

4.2 Assembly Code

No optimization or special flags were used when generating assembly code. Lines of assembly code were counted using the vim editor based only on the assembly files generated from the input C++ files and not from the OpenC++ metafiles. Furthermore, all assembly code comes from the compilation of the *.ii files generated by OpenC++, which contain the modified input code along with OpenC++ specific code. When all OpenC++ specific code is removed, the resulting assembly file is much smaller. For distribution, it is only 82 lines compared with 269 before removal. It is assumed that the OpenC++ specific code contains links to OpenC++ libraries. Interestingly, when the solution is hard-coded without the use of metaprogramming or template programming, the assembly code is larger. For instance, the assembly code for distribution is 274 lines long.

4.2.1 Distribution (One-way)

With only 269 lines of code, one-way distribution is the shortest solution. All assembly code is located in a single file generated from the Tester class and features only three functions, bind_c_1(), tweak(), and the main() itself. There are only three function calls, two to tweak() and one to bind_c_1(), and a single programming structure, the for loop. The binding of the components is done through a direct pointer negating the need for inheritance, extra objects, or explicit invocation. Please see Appendix C for the annotated assembly code.

4.2.2 Status Variable (One-way)

The one-way status variable contains additional code for the Updater class inherited by F resulting in 384 lines of code. Also, the update() method is located in a separate file and requires an extra jump for access. It alone adds 61 lines of code. Another source of extra code is the code required to pass the pointer of C to F in order for F to retrieve the new value of c.

4.2.3 Status Variable (Two-way)

Two-way status variable code is based on the one-way status variable code except both C an F have the tweak(), bind(), and update() methods. This is responsible for the increase of code along with an extra call to bind() present in the main() method resulting in 512 lines. This is the largest metaprogramming solution.

4.2.4 Pointer (One-way)

The code of one-way pointer is greatly simplified from the one-way status variable. The Updater class code is no longer present negating the need for F to inherit it and resulting in 289 lines.

4.2.5 Pointer (Two-way)

The increase in the amount of assembly code to 378 comes from duplicating the bind(), update(), and tweak() methods just like in two-way status variable. However, lack of Updaters class allows a great reduction in assembly code in comparison to two-way status variables at the cost of explicit invocation.

4.2.6 Inheritance (One-way)

The inheritance of C by F within the one-way status variable design simply added the code from C into F resulting in larger total line count of 418. All other assembly code remained same as in one-way status variable.

4.2.7 Mediator (Two-way)

The mediator solution resulted in an assembly code significantly different from all other solutions. First, there is a new Mediator class that encapsulates the pointers to C and F. It also contains the invariant maintenance code for the two-way invariant within the update() methods. Unlike the status variable solution where the dependent component gets the value of the independent component and employs it in calculating its new value, the mediator handles the calculation and just updates the dependent component with the new value. However, the components must have the tweak() method to alert the mediator to the invariant change. While this increases the size of the assembly code, the new tweak() accepts the new component value and updates the component while alerting the mediator. This allows for the removal of the assignment in main() generating 437 lines of assembly code.

4.2.8 Mediator (Optimized) (Two-way)

The mediator optimized solution contains the invariant maintenance code for the two-way invariant within the tweak() methods. Instead of having the independent component alert the mediator to the change, the optimized solution allows the Tester to do it directly. This way, the independent component has no knowledge of the mediator and no tweak method() This results in much simpler communication code since all communication between components is one way at each update. Furthermore, because the mediator contains all binding and invariant code, the C and F classes contain just their respective values. As a result, the mediator is the shortest and most streamlined of the two-way solutions with only 304 lines of code.

4.2.9 Template Status Variable (One-way)

The longest assembly code is produced by the template programming solution with 559 lines. It is divided among three files and includes code for eight different classes as shown in **Figure 15**. Through templates, the compiler is able to determine the type of the independent variable at compile time and place references to it in the code. Nevertheless, extra code occurs in the wrapper classes C_Top and B_Bot. This code includes not only the methods these classes contain but also the inheritance of C and F. As shown by the one-way inheritance solution, inheritance by itself carries cost in extra assembly code. Moreover, the template solution features extra code from inheritance of Updaters, StatusVariable, and SVC classes. However, due to the nature of templates, code for assignment override was not transferred over into the assembly code but used to change the assignment at compile time. In conclusion, template programming allows optimization within the assembly code by determining certain calculations at compile time, but the overall designed used for the template status variable solution results in the largest assembly code of all solutions due to increased wrapping and inheritance.

4.3 Summary

All nine solutions were tested on a single system under identical conditions. With runtime, and size of the executable being virtually the same, the differences between solutions are displayed in

compile time and assembly code. Metaprogramming solutions resulted in shorter assembly code with distribution and optimized mediator being the shortest for one-way and two-way invariants respectively. Template programming excelled at short compile time but produced the largest and most complex assembly code.

5 Discussion

5.1 Implications

5.1.1 Quantitative Analysis

From the data gathered during the case study, the advantages of metaprogramming lie in the assembly code while template programming has a shorter time to compile. Template status variable results in assembly code almost 50 lines longer than the longest metaprogramming solution. What is more, the metaprogramming solution is two-way while the template solution is only one-way. The smallest one and two-way metaprogramming solutions, distribution and mediator respectively, are almost half the size of the template solution. Lastly, the one-way status variable metaprogramming solution, while based on the design of the template solution, is almost 180 lines smaller.

Conversely, metaprogramming compile times are 15 to 17 times as long as using template programming. It should be noted that templates usually take longer to compile in comparison to C++ code that does not include them. The difference is probably due to templates being a part of the GNU compiler while metaprogramming comes as a separate tool requiring multiple steps for translating from metacode to executable. Although not as important as runtime for the end user, long compile time can complicate development and testing for very large and complex system.

The results for runtime were virtually the same for all solutions. The only standouts were two-way status variable and pointer metaprogramming which were from one-sixth to one-third longer. However, different time results appeared for every trial and all solutions were constantly between 0.030 and 0.046 seconds. Thus, it is safe to conclude that runtime for all solutions was the same. Perhaps, the testing code dominated so the differences were not apparent in the approaches. Otherwise, the problem could be too simple to differentiate performance.

5.1.2 Qualitative Analysis

The results of qualitative analysis have been gathered during the entire software development process and include the ease of installation, the ability of each technique to solve the implicit invocation of invariant maintenance problem, the ease of adjusting input code for the technique, and how well the technique fits-in with the developer's process of thinking. Serious problems occurred when trying to install and use OpenC++ for metaprogramming. For their complete description please see **Appendix A.** On the other hand, since templates are part of the GNU C++ compiler, there were absolutely no difficulties in using them. It should be noted that the problems encountered were with the specific tool, OpenC++, and not with the technique of metaprogramming. With more development, there is no reason to doubt that OpenC++ or another tool could be as easy to use as C++ templates.

Once OpenC++ began functioning, it was used to implement, the eight different metaprogramming solutions. All were done with relative ease and much faster than was originally planned. In fact, most solutions were simply redesigns of the very first metaprogramming solution involving pointers. This allowed one-way, two-way, and unusual designs like mediator and distribution to be implemented quickly and easily. Most importantly, the ability to see the code before and after metaprogramming combination allowed for fast and easy debugging. However, the metacode itself was much harder to debug due to cryptic error messages and lack of up-to-date documentation.

In opposition, the layered design of the template programming solution was complex and hard to understand. Multiple readings of the DYNAMO project research papers were required just to realize the purpose of templates in the solution. Furthermore, a two-way solution is impossible to implement with the nested approach due to the order of layers. It is hard to imagine how template programming and mixin layers could be used to implement other approaches such as mediator, distribution, or aggregation. Finally, templates in C++ do not support clear error messages and debugging the design was difficult.

If the two techniques are to be implemented automatically upon an input source code, the changes required in that code needs to be evaluated. Each metaprogramming solution requires an additional one to two lines of code within the main() method while mostly keeping the previous declaration of components intact. Template programming requires two new lines to replace the component declarations. No metaprogramming solution requires more than one line to be replaced. Consequently, metaprogramming allows fewer and easier changes to the input code.

Last important criterion of qualitative analysis is how well each technique matches the developer's thinking process. Metaprogramming excels at this because the input and the output code are relatively simple allowing each metaprogramming solution to be hard-coded in C++ before going through OpenC++. Each hard-coded solution can be tested and analyzed at will to determine the best design. Afterwards, the solution is divided into input code and metaprogramming generated code. The later is removed and inserted into OpenC++'s metacode for automatic generation. As a result, the developer is always thinking in terms of plain C++ code.

In template programming, the developer must decide on how to use templates to complete the task. DYNAMO features just one possible solution and it needs to be adjusted for all but the most basic invariants. Unless the developer is experienced in template programming and template theory, this task could prove extremely difficult. In summary, metaprogramming makes due with knowledge of basic C++ whereas template programming requires a different approach to the problem with concrete understanding of templates.

5.2 Future Work

Due to time constraints, only one instance of template programming was implemented. Mediator and distribution versions of template programming must be implemented to attain complete comparison. Furthermore, no attempt was made to create two-sided invariants using template programming. Doing this would help to quantify exactly how difficult such task is. Finally,

while the invariant implemented granted good data, it was too simple to gain meaningful runtime comparisons. Implementing a more complex example with multiple components and invariants in a single system would permit for better analysis of runtime and architecture.

6 Conclusion/Reflections

The results of the case study lead to the conclusion that metaprogramming is superior to template programming for the implementation of implicit invocation of invariant maintenance due to smaller final assembly code, superior ability to solve the problem with multiple approaches and two-way invariants, and easier fit into the developer's mindset. The only areas where metaprogramming lacks behind template programming are compile time and ease of installation. The first area is not of particular importance to the final user while the second area is based solely on the negative experience with OpenC++ as documented in **Appendix A**. If C++ metaprogramming continues to evolve, it should overcome both defects. Although C++ templates can be improved with better error messages and faster compile times, their Turing-complete nature will always keep template specialization too complex in comparison to non-template metaprogramming.

Reflecting upon completed work, it is clear that much has been learned. First, I finally understood exactly what metaprogramming is and how it works. While I was introduced to it in previous courses, actually using it showed me a new dimension of programming. In particular, I learned about different types of metaprogramming such as runtime reflection and compile time metaprogramming. This helped me to compare and contract computer languages like Smalltalk and C++. Also, I learned about how metaprogramming is implemented within different languages through the use of the interpreter or a separate program akin to OpenC++.

In addition to learning about metaprogramming, I gained better understanding of approaches to invariant maintenance. Although I previously studied how distribution, mediators, and implicit invocation worked, implementing them and seeing the implementation in assembly helped me to comprehend their importance within larger systems. For example, using pointers to connect components may seem easy, but it results in entangled and untraceable code within a complex system. On the other hand, the mediator approach leaves all original components completely free of changes and places all invariant maintenance code in a single accessible location.

Finally, I gained important insights into research projects and tools. Specifically, far too much time was spent trying to fix OpenC++. Instead, there should have been a contingency plan in place to be executed if OpenC++ did not function within a certain time period. I should also have tested and assessed OpenC++ before deciding to use it for the project. Likewise, I should have searched for other ways of doing metaprogramming in C++. On the other hand, the experience with OpenC++ taught me what metaprogramming and metacode look like at different stages of implementation. I was able to view the abstract syntax tree that contained my C++ code and modify in through a various means. If another tool hid this while functioning correctly from the start, I would have completed more work but not learn as much about the metaprogramming and its implementation.

7 Acknowledgements

I would like to extend my deepest appreciation to Scott D. Fleming of Michigan State University for help in getting OpenC++ to run and to Dr. Kurt Stirewalt of Michigan State University for providing examples on OpenC++ metacode for invariant maintenance. I would also like to thank all members of the OpenC++ community for advice and assistance in using OpenC++. Finally, special recognition goes to Dr. Spencer Rugaber of Georgia Institute of Technology without whose guidance, this research would not be possible.

8 Works Cited

[1] D. Garlan and M. Shaw. An Introduction to software architecture. In Advances in Software Engineering and Knowledge Engineering, pages 1-39, Singapore, 1993. World Scientific Publishing Company.

[2] Czarnecki, Krzysztof and Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[3] Object Management Group, "Object Constraint Language Specification". OMG Unified Modeling Language Specification, 1.5. Chapter 6, March 3, 1999.

[4] I. R. Forman, S. H. Danforth. Putting Metaclasses to Work. Addison-Wesley 1999.

[5] David Vandevoorde, C++ Templates: The Complete Guide, 2003 Addison-Wesley.

[6] Mehmet Aksit , Ken Wakita , Jan Bosch , Lodewijk Bergmans , Akinori Yonezawa, "Abstracting Object Interactions Using Composition Filters," Proceedings of the Workshop on Object-Based Distributed Programming, p.152-184, July 26-27, 1993.

[7] Robert DeLine. "Avoiding packaging mismatch with Flexible Packaging." *IEEE Transactions on Software Engineering* 27(2):124-143, February 2001.

[8] Sullivan, K.J. and D. Notkin, "Reconciling Environment Integration and Software Evolution" *ACM Transactions on Software Engineering and Methodology,* vol. 1, no. 3., pp. 229-268, July, 1992.

[9] Kevin J. Sullivan , Ira J. Kalet , David Notkin, "Evaluating The Mediator Method: Prism as a Case Study", IEEE Transactions on Software Engineering, v.22 n.8, p.563-579, August 1996.

[10] Foote, Brian, and Ralph E. Johnson. "Reflective Facilities in Smalltalk-80." Brian Foote. 16 Oct. 1989. Dept. of Computer Science, University of Illinois at Urbana-Champaign. 8 Oct. 2005 <http://www.laputan.org/ref89/ref89.html>.

[11] Fred Rivard. "Smalltalk: a Reflective Language". In *Reflection'96,* April 1996.

[12] Dale, Green. "Trail: The Reflection API." Sun Microsystems. 18 Oct. 2005 <http://java.sun.com/docs/books/tutorial/reflect/>.

[13] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. *OpenJava: A class-based macro system for Java.* In W. Cazzola, R. J. Stroud, and F. Tisato, editors, Reflection and Software Engineering, LNCS 1826, pages 119--135. Springer-Verlag, July 2000.

[14] Knizhnik, Konstantin. "Reflection for C++." The Garret Group. 4 Nov. 2005 <http://www.garret.ru/~knizhnik/cppreflection/docs/reflect.html>.

[15] Roiser, Stefan. "Reflection in C++." CERN EP/LBC, TU Vienna. December 15, 2003.

[16] Vollmann, Detlef. "Metaclasses and Reflection in C++." Vollmann Engineering. 2000. 27 Nov. 2005 <http://www.vollmann.com/pubs/meta/meta/meta.html>.

[17] Chiba, Shigeru. "A Metaobject Protocol for C++." In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), page 285-299, October 1995.

[18] G. Kiczales. "Towards a new model of abstraction in software engineering." In Proc. of IMSA'92 Workshop on Reflection and Meta-level Architectures, 1992.

[19] Fleming, Scott D., Betty H.C. Cheng, R.E. Kurt Stirewalt, and Philip K. McKinley. "An approach to implementing dynamic adaptation in C++." In Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005 (DEAS'05), in conjunction with ICSE 2005, St. Louis, Missouri, May 2005.

[20] D. Batory and B. J. Geraci. "Composition validation and subjectivity in GenVoca generators." IEEE Transactions on Software Engineering, pages 67–82, Feb. 1997.

[21] Y. Smaragdakis and D. Batory. "Implementing Layered Designs with Mixin Layers." Proceedings of the 12th European Conference on Object-oriented Programming, 1998.

[22] Rugaber, Spencer and Stirewalt, Kurt. "Metaprogramming Compilation of Invariant Maintenance Wrappers from OCL Constraints." College of Computing, Georgia Institute of Technology Technical Report: GIT-CC-03-46. October 27, 2003.

[23] Rugaber, Spencer and Stirewalt, Kurt. "Final Project Report: Dynamic Assembly from Models (DYNAMO)." College of Computing, Georgia Institute of Technology. October 30, 2003.

[24] "DYNAMO Design Guidebook." College of Computing, Georgia Institute of Technology Technical Report: GIT-CC-01-37. June 27, 2002.

9 Appendix A – OpenC++ Evaluation

9.1 Advantages

9.1.1 Metaprogramming implemented quickly and easily

OpenC++ provides an API that allows fast access to and modification of C++ source code. For example, specifying what variable name to search for and what to do once it is found takes only a few lines of code. Similarly, the abstract syntax tree that holds parsed code provides information about each code structure like its name and scope.

9.1.2 Available documentation

Full documentation for OpenC++ API is available on the project website (http://opencxx.sourceforge.net/) along with research papers that explain how OpenC++ works. Some research papers also contain code examples and their analyses.

9.1.3 Available assistance on the newsgroup

The project newsgroup, linked from the project website, allow access to assistance from more experienced OpenC++ users. Experience showed that all posts are replied to within 24 hours and usually by multiple people. The wealth of knowledge and experience on the newsgroup is much greater in comparison to all on-line sources combined.

9.1.4 Used within academia and industry

Unlike most freeware tools, OpenC++ is expensively employed within academia. Georgia Institute of Technology uses it to tech object-oriented systems and languages while Michigan State University uses it in conducting research. Additionally, Debian and Linspire, two popular Linux distributions, released OpenC++ packages for their respective systems.

9.1.5 Code done in regular C++

Instead of requiring the user to learn a new language specifically designed for the tool, OpenC++ metacode is written using regular C++. This reduces the learning curve and allows the user to combine metacode with existing C++ code in order to create new tools.

9.2 Disadvantages

9.2.1 Available version does not work properly

Although OpenC++ is easy to learn and use once it is properly installed, getting to this step requires a large amount of unnecessary work. First of all, the tool is very fastidious about its environment such as requiring a specific version of gcc and libtool libraries. Likewise, the current distribution contains major defects that must be fixed before it can be installed. The information about the defects can be found on the newsgroup but not on the project webpage. Even after all requirements have been satisfied and the system is installed, it is unable to compile all but the most basic metaprograms. To make it fully functional, complete recompilation on the local system is required. Otherwise, the recompiled code must be received from a third-party source. Finally, versions available in Debian and Linspire distributions are completely non-

functional. If this information was available on the project website, users could estimate the time requirements for their project. However, the project webpage only contains a link for reporting bugs but no mention of existing defects or their status.

9.2.2 Documentation is out-of-date

While the project website contains a complete and well-documented API, it is severely out-of-date. Many functions do not exist within the API but can be found on other websites (http://www.csg.is.titech.ac.jp/~chiba/openc++.html) and papers. Likewise, most research papers based on OpenC++ are fairly old and contain code that no longer works.

9.2.3 No comprehensive source of help or examples

Throughout its existence, OpenC++ was used in many projects and is mentioned in multiple research papers. However, aside from Shigeru Chiba's website, there is no comprehensive source of past and current OpenC+ research. Only a few research papers feature examples of metacode and explanations about its functionality. Moreover, all examples are severely outdated or are too simple to be useful.

9.2.4 Compilation time

Programs that use OpenC++ for metaprogramming require much longer to compile in comparison to regular C++ or C++ template programming due to the number of steps taken before the actual compilation. Those steps are described in the **Approach** section. Unlike runtime, compile time is not noticeable to the end user of the application; therefore, it is not as important then compared to other OpenC++ disadvantages.

9.3 Conclusion

It is hard to conclude about the usefulness of OpenC++ and its future. The current implementation offers great functionality and is undeniably beneficial in C++ metaprogramming but only when it can be satisfactorily installed. However, current steps required to make OpenC++ operational are too complex for most non-experts. Perhaps OpenC++ should be combined with gcc to improve compatibly and increase distribution. This would require major bug fixes but no changes in the overall tool design.

10 Appendix B – Template Programming Evaluation

10.1 Advantages

10.1.1 Integral part on C++

Templates are an integral part of C++ supported by most C++ compilers including GNU gcc. They require no additional installation or configuration.

10.1.2 Available documentation

There are thousands of books, magazines, articles, web-pages, and research papers written about C++ templates and their uses including works by the author of C++, Bjarne Stroustrup.

10.1.3 Available assistance

Since templates are widely available and used, there are thousands of experts, newsgroups, and help sites devoted to their use.

10.2 Disadvantages

10.2.1 Not specialized enough

Given that C++ templates are Turing-complete, they can be used to implement any possible program. Often, they are used for collection classes but they can also be employed in metaprogramming. This can make it hard to figure out how templates can be used to solve a specific problem like invariant maintenance. The DYNAMO project offers one type of solution but there could be others.

10.2.2 Poor error messages

When errors occur at compile due to incorrectness of templates, the error message can be cryptic.

10.3 Conclusion

As it stands today, C++ templates are more mature in comparison to a third party tool like OpenC++. They are well documented and have plenty of available resources for assistance. However, specialized metaprogramming tools provide a more logical solution to the implementation of implicit invocation for invariant maintenance. After the initial cost of installing and learning the tool, metaprogramming is faster to implement, provides wider architectural support al lowest cost, and presents a more natural way to think about the problem.

11 Appendix C – Distribution Assembly Code

```
	.file	"tester.cc"
	.text
	.align 2
	.def	__ZSt17__verify_groupingPKcjRKSs;	.scl	3;	.type	32;	.endef
__ZSt17__verify_groupingPKcjRKSs:
	pushl	%ebp
	movl	%esp, %ebp
	subl	$40, %esp
	movl	16(%ebp), %eax
	movl	%eax, (%esp)
	call	__ZNKSs4sizeEv
	decl	%eax
	movl	%eax, -4(%ebp)
	movl	12(%ebp), %eax
	decl	%eax
	movl	%eax, -12(%ebp)
	leal	-12(%ebp), %eax
	movl	%eax, 4(%esp)
	leal	-4(%ebp), %eax
	movl	%eax, (%esp)
	call	__ZSt3minIjERKT_S2_S2_
	movl	(%eax), %eax
	movl	%eax, -8(%ebp)
	movl	-4(%ebp), %eax
	movl	%eax, -16(%ebp)
	movb	$1, -17(%ebp)
	movl	$0, -24(%ebp)
L2:
	movl	-24(%ebp), %eax
	cmpl	-8(%ebp), %eax
	jae	L5
	cmpb	$0, -17(%ebp)
	je	L5
	movl	-16(%ebp), %eax
	movl	%eax, 4(%esp)
	movl	16(%ebp), %eax
	movl	%eax, (%esp)
	call	__ZNKSsixEj
	movl	%eax, %ecx
	movl	-24(%ebp), %eax
	movl	8(%ebp), %edx
	addl	%eax, %edx
	movzbl(%ecx), %eax
	cmpb	(%edx), %al
```

```
        sete    %al
        movb    %al, -17(%ebp)
        leal    -16(%ebp), %eax
        decl    (%eax)
        leal    -24(%ebp), %eax
        incl    (%eax)
        jmp     L2
L5:
        cmpl    $0, -16(%ebp)
        je      L6
        cmpb    $0, -17(%ebp)
        je      L6
        movl    -16(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    16(%ebp), %eax
        movl    %eax, (%esp)
        call    __ZNKSsixEj
        movl    %eax, %ecx
        movl    -8(%ebp), %eax
        movl    8(%ebp), %edx
        addl    %eax, %edx
        movzbl(%ecx), %eax
        cmpb    (%edx), %al
        sete    %al
        movb    %al, -17(%ebp)
        leal    -16(%ebp), %eax
        decl    (%eax)
        jmp     L5
L6:
        movl    $0, 4(%esp)
        movl    16(%ebp), %eax
        movl    %eax, (%esp)
        call    __ZNKSsixEj
        movl    %eax, %ecx
        movl    -8(%ebp), %eax
        movl    8(%ebp), %edx
        addl    %eax, %edx
        movzbl(%ecx), %eax
        cmpb    (%edx), %al
        jg      L8
        movzbl-17(%ebp), %eax
        andl    $1, %eax
        movb    %al, -25(%ebp)
        jmp     L9
L8:
        movb    $0, -25(%ebp)
```

```
L9:
        movzbl-25(%ebp), %eax
        movb   %al, -17(%ebp)
        movzbl-17(%ebp), %eax
        leave
        ret
.lcomm __ZSt8__ioinit,16
        .def    ___main;        .scl    2;      .type   32;     .endef
```
Change in the value of c
```
        .section .rdata,"dr"
        .align 8
LC1:
        .long   0
        .long   1079574528
        .text
```
End of change in the value of c
```
        .align 2
.globl _main
        .def    _main; .scl     2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $72, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        movl    %eax, -44(%ebp)
        movl    -44(%ebp), %eax
        call    __alloca
        call    ___main
        leal    -32(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    -24(%ebp), %eax
        movl    %eax, (%esp)
```
Binding on the instance of the F class to the instance of the C class through call to bind_c_1
```
        call    __ZN1C8bind_c_1EP1F
        movl    $0, -36(%ebp)
L11:
```
Start of the 10,000 loop
```
        cmpl    $9999, -36(%ebp)
        jg      L12
```
First call to tweak
```
        fldz
```

```
        fstpl   -24(%ebp)
        leal    -24(%ebp), %eax
        movl    %eax, (%esp)
        call    __ZN1C5tweakEv
End of first call to tweak
Second call to tweak
        fldl    LC1
        fstpl   -24(%ebp)
        leal    -24(%ebp), %eax
        movl    %eax, (%esp)
        call    __ZN1C5tweakEv
End of second call to tweak
        leal    -36(%ebp), %eax
        incl    (%eax)
        jmp     L11
End of loop
L12:
        movl    $0, %eax
        leave
        ret
        .section .rdata,"dr"
        .align 8
LC3:
Inside the tweak method on C
The invariant rule
        .long   -858993459
        .long   1073532108
        .align 8
LC4:
        .long   0
        .long   1077936128
        .section        .text$_ZN1C5tweakEv,"x"
        .linkonce discard
        .align 2
End of the invariant rule
.globl __ZN1C5tweakEv
        .def    __ZN1C5tweakEv;   .scl    2;      .type   32;     .endef
__ZN1C5tweakEv:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    8(%eax), %edx
        movl    8(%ebp), %eax
        fldl    (%eax)
        fldl    LC3
        fmulp   %st, %st(1)
```

```
        fldl    LC4
        faddp   %st, %st(1)
        fstpl   (%edx)
        popl    %ebp
        ret
        .section        .text$_ZN1C8bind_c_1EP1F,"x"
        .linkonce discard
        .align 2
```

The bind method that binds the instance of F to the instance of C

```
.globl __ZN1C8bind_c_1EP1F
        .def    __ZN1C8bind_c_1EP1F;    .scl    2;      .type   32;     .endef
__ZN1C8bind_c_1EP1F:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        movl    %eax, 8(%edx)
        popl    %ebp
        ret
        .section        .text$_ZSt3minIjERKT_S2_S2_,"x"
        .linkonce discard
        .align 2
```

End of the bind method

```
.globl __ZSt3minIjERKT_S2_S2_
        .def    __ZSt3minIjERKT_S2_S2_; .scl    2;      .type   32;     .endef
__ZSt3minIjERKT_S2_S2_:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    12(%ebp), %eax
        movl    8(%ebp), %edx
        movl    (%eax), %eax
        cmpl    (%edx), %eax
        jae     L17
        movl    12(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     L16
L17:
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
L16:
        movl    -4(%ebp), %eax
        leave
        ret
        .text
        .align 2
```

```
        .def    __Z41__static_initialization_and_destruction_0ii;  .scl    3;      .type   32;
        .endef
__Z41__static_initialization_and_destruction_0ii:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        cmpl    $65535, 12(%ebp)
        jne     L19
        cmpl    $1, 8(%ebp)
        jne     L19
        movl    $__ZSt8__ioinit, (%esp)
        call    __ZNSt8ios_base4InitC1Ev
L19:
        cmpl    $65535, 12(%ebp)
        jne     L18
        cmpl    $0, 8(%ebp)
        jne     L18
        movl    $__ZSt8__ioinit, (%esp)
        call    __ZNSt8ios_base4InitD1Ev
L18:
        leave
        ret
        .align 2
        .def    __GLOBAL__I_main;       .scl    3;      .type   32;     .endef
__GLOBAL__I_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $65535, 4(%esp)
        movl    $1, (%esp)
        call    __Z41__static_initialization_and_destruction_0ii
        leave
        ret
        .section        .ctors,"w"
        .align 4
        .long   __GLOBAL__I_main
        .text
        .align 2
        .def    __GLOBAL__D_main;       .scl    3;      .type   32;     .endef
__GLOBAL__D_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $65535, 4(%esp)
        movl    $0, (%esp)
        call    __Z41__static_initialization_and_destruction_0ii
```

```
leave
ret
.section        .dtors,"w"
.align 4
.long   __GLOBAL__D_main
.def    __ZNSt8ios_base4InitD1Ev; .scl    3;      .type   32;     .endef
.def    __ZNSt8ios_base4InitC1Ev; .scl    3;      .type   32;     .endef
.def    __ZNKSsixEj;.scl    3;      .type   32;     .endef
.def    __ZNKSs4sizeEv;     .scl    3;      .type   32;     .endef
.def    __ZSt3minIjERKT_S2_S2_; .scl    3;      .type   32;     .endef
```