

# The Design of an OCL Query-Based Debugger for C++

Chanika Hobatr  
Clemson University  
Computer Science Dept  
Clemson, SC USA  
puh@cs.clemson.edu

Brian A. Malloy  
Clemson University  
Computer Science Dept  
Clemson, SC USA  
malloy@cs.clemson.edu

## ABSTRACT

In this paper, we describe an approach to query-based debugging for C++ programs that uses queries formulated in the object constraint language, OCL. By using OCL, together with UML, to formulate queries, debugging can begin early in the design phase. Queries can be formulated to verify design constraints such as invariants on classes and pre and post-conditions for member functions. These queries can be reused after code generation to verify the design contract, as part of the testing process, and to facilitate fault detection.

## General Terms

Debugging, query, code instrumentation, code generation, OpenC++, Unified Modeling Language (UML), Object Constraint Language (OCL), meta-class Meta-object Protocol (MOP).

## 1. INTRODUCTION

The task of debugging programs has grown increasingly difficult with the increased size and complexity of software applications. The acceptance of object technology as the paradigm of choice for software development has produced programs with stronger cohesion and encapsulation and has facilitated the task of modifying and extending code. However, object technology has also increased the difficulty of the debugging task due to the burgeoning number of objects generated in the program. The debugging task is difficult for programs written in object-oriented languages such as Java, even though the Java memory model is straightforward: all objects are placed on the heap. The debugging task for C++ programs is more difficult than Java since the C++ memory model permits the generation of objects for both the stack and the heap.

A further difficulty in debugging is the delay in appearance of the bug itself. This cause-effect gap between the place in the code where the error is **introduced** and the place in the code where the error **occurs** can make debugging a daunting

task even for the experienced programmer[?]. For example, assume that a C++ programmer fails to allocate memory for an object that is typically allocated on the stack, but is occasionally placed on the heap. This error may not appear until much later in the execution than the place where the heap allocation failed to occur. To facilitate the debugging process, traditional debuggers focus on source code lines, permitting the user to apply break points in the code where an error is likely to occur. The cause-effect gap nullifies the power of these traditional debuggers.

A final difficulty in debugging relates to a search for errors that are introduced by an incorrect interpretation of the design of the software. For example, consider a stack class with an invariant that requires that the pointer to the top element is never negative. An occurrence of this error may produce spurious results that are difficult for the programmer, sometimes far removed from the design, to detect.

A recent approach for addressing the increasing difficulty of debugging object-oriented software that is more promising than the traditional breakpoint approach is query-based debugging[?; ?; ?]. In this approach, the programmer may introduce either static or dynamic queries into the system in an effort to detect the offending object. The debugger monitors the system, evaluating the queries as execution progresses. The queries permit the programmer to examine object relationships at runtime. The difficulty with these query-based approaches is the placement of undue burden on the programmer to ensure that queries are side-effect free. The problem in using queries with side-effects is that the query itself may introduce an error into the program. A further difficulty in previous approaches is that the queries are formulated in the underlying implementation language[?; ?; ?]. This approach does not facilitate formulation of queries at the design stage of software development.

In this paper, we describe an approach to query-based debugging for C++ programs that uses queries formulated in the object constraint language, OCL[?]. By using OCL, together with UML, to formulate queries, debugging can begin early in the design phase. Queries can be formulated to verify design constraints such as class invariants and pre and post-conditions for member functions[?]. These queries can be reused after code generation to verify the design contract, as part of the testing process, and to facilitate fault detection.

A further advantage of using OCL to formulate queries is that it is an expression language and, as such, is side-effect free. Previous approaches to query-based debugging assume that method calls are side-effect free[?; ?]. In our approach,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2001 Las Vegas, USA.

Copyright 2001 ACM 0-89791-88-6/97/05 ..\$5.00

we only permit OCL function calls to C++ const member functions. Our implementation exploits the meta-object protocol of OpenC++[?].

The remainder of this paper is organized as follows. In the next section we provide background about the meta-object protocol, OpenC++, and about the object constraint language that we use to formulate queries. In Section ?? we present the design of our OCL query-based debugger, OQBD. In Section ?? we draw conclusions.

## 2. BACKGROUND

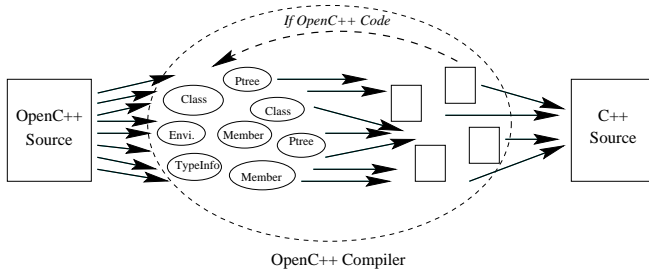


Figure 1: *The OpenC++ Meta-object Protocol.*

In this section we provide background about OpenC++, the meta-object protocol that forms the basis of our implementation of OQBD. In Section ?? we overview the object constraint language, OCL, that we use to formulate queries in our system.

### 2.1 OpenC++

OpenC++ is a meta-object protocol, MOP, for extending the C++ programming language[?; ?]. The MOP choreograph source to source translation from OpenC++ to C++, allowing developers to extend the language to include structures such as persistence, distributed objects or customized compiler optimizations. The MOP gives developers greater control over the compilation of programs by facilitating the extension of class definitions, member function access, virtual function invocation, and object creation. Some examples of the extensions provided by the MOP of OpenC++ include the unobtrusive monitoring of class functions and data attributes, optimizations for a matrix library and easy application of wrapper functions[?]. A unique feature of OpenC++ is that it permits users to extend C++ language features with minimal modification of the source code. In addition, since the protocol is applied statically rather than dynamically, the OpenC++ model is efficient enough for “real world” programs[?].

The protocol of OpenC++ is summarized in Figure ??, where the left side of the figure illustrates the input to the system and the right side of the figure illustrates the output of the system. Input to the OpenC++ system, as shown in the figure, is OpenC++ source, a combination of C++ code and MOP directives to accomplish the extensions to C++ desired by the MOP user.

This OpenC++ source is partitioned into top-level class definitions and member functions. A meta-object is constructed for each such definition and this meta-object translates the top-level definition into ISO C++ code[?]. A meta-object represents meta-aspects of the C++ language, as described in the previous paragraph. Figure ?? shows the constructed

meta-objects, listing them as Class, Environment, Typeinfo, Ptree and Member objects. By manipulating these meta-objects, programmers can control source-to-source translation from OpenC++ to C++. A summary of the functionality of the meta-objects follows:

- Class meta-objects represent class definition and control source-to-source translation of the program via inheritance and virtual functions.
- Ptree meta-objects represent parse trees of a program text. It is implemented as a nested linked-list of lexical tokens.
- Member meta-objects represent class members, i.e. data members and function members, and contain information about those members.
- TypeInfo meta-objects represent types.
- Environment meta-objects represent bindings between names and types.

Each meta-object takes responsibility for translating each top-level definition into appropriate C++; this translation is illustrated in Figure ?? by the arrows from the ellipses, in the central left of the figure, to the rectangles in the central right of the figure. The dotted arrow moving from the rectangles back to the meta-objects (right to left in the figure) illustrates the significant point that meta-objects are permitted to generate arbitrary code that may contain OpenC++ code, and thus will require further translation.

To extend the functionality of C++ to include MOP, the user must declare a meta-class for each class to be extended by prefixing the class definition with an OpenC++ directive, as illustrated in the following example:

```
metaclass PostCondClass Point;
class Point {
public:
    int x, y;
    Move(int inx, int iny);
};
```

In the code segment above, the annotation before Point indicates that the base-level class Point is to become an instance of the metaclass PostCondClass. Translation of Point is controlled by PostCondClass, implemented as a meta-level program. Only classes that are annotated with OpenC++ directives are translated to include the meta-object protocol. For our example, those classes include Point and any classes in Points inheritance hierarchy. Classes that are not OpenC++ annotated are unaffected.

The translation of OpenC++ programs proceeds in three phases: a preprocessor phase to construct meta-objects, translation of each meta-object from OpenC++ to regular C++, and a back-end C++ compiler.

### 2.2 OCL

A branch of computer science that is concerned with using formal logical languages to give precise and unambiguous descriptions of specifications of programs is becoming increasingly popular. This branch has been historically referred to as *formal methods* and they have produced both interesting and informative research using languages such as **Z**, *VDM* and Larch.

For object-oriented development of applications, the recent arrival of the *unified modeling language*, UML, has allowed many developers to find a common language for modeling the specification, design and state changes of an object-oriented system. However, there are many subtleties and nuances in the design and specification of systems that the UML is not able to capture. To address this UML shortcoming, the object constraint language, OCL, has been designed as part of the Object Management Group effort[?]. OCL has been used to give additional precision to the definition of UML; also, OCL has become part of the UML standard to permit modelers to specify additional nuances in their UML documentation.

OCL is an expression language designed to function with UML to enable constraint expression for pre and post conditions, class invariants, guards and any kind of constraint over the objects in a system. An OCL expression is guaranteed to be without side effect. It cannot change anything in the model. Thus, the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify such a state change (for example, in a post-condition). All values for all objects, including all links, will not change. When an OCL expression is evaluated, it simply delivers a value[?].

OCL is a modeling language, not a programming language. It is not possible to write program logic or flow-control in OCL. You especially cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, not everything in it is promised to be directly executable[?].

### 3. DESIGN OF THE DEBUGGER

The OQBD system is composed of two disparate subsystems that we refer to as a *front-end* and a *back-end*, illustrated in Figure ?? . The front-end is an augmented OCL compiler that accepts an OCL query and generates C++ code together with the query information necessary for debugging; the generated code is placed in an output file for further processing. This output file includes the name of the class under consideration, the type of constraint (pre-condition, post-condition, or class invariant), the name and signature of a function if the constraint is a pre-condition or post-condition, and the list of attributes if the constraint is a class invariant. Finally, the output file includes the C++ code that is generated to represent the constraint.

The back-end reads the information from the front-end and instruments it into the pre-compiled base-level C++ source code that will be debugged. The instrumented C++ source code is again compiled by the C++ compiler. The output from the back-end is an executable program with constraints specified in the OCL query plus runtime support code called the QueryEvaluator to evaluate queries during the program execution if needed.

#### 3.1 The Front-end

The front-end is an OCL compiler that parses and generates C++ instructions from an OCL constraint or query. We have modified a version of the OCL compiler developed by Dresden Technology University[?] to permit us to generate C++ code instead of Java code. A construction of the compiler [?] consisting of a parser, a semantic analyzer, a normalizer, and a code generator is depicted in Figure ?? .

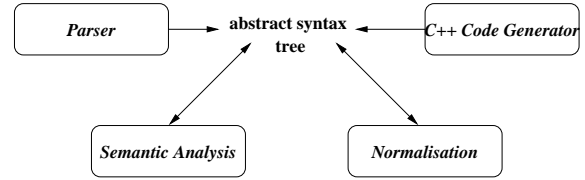


Figure 3: OCL Compiler construction.

To generate C++ statements from OCL queries, first, the OCL queries that are given into the debugger as constraints of the program are checked to see if they are syntactically correct and are well-formed according to the OCL language. The queries are parsed by the parser, built using the tool SableCC, with reference to the UML meta-model of the program. Next the parsed queries in the form of an abstract syntax tree is semantically analyzed and normalized before the final step of code generation [?].

The C++ code generator is the part of the OCL compiler that generates C++ code from the given queries including any corresponding information required for debugging in the next phase, such as, classname, action (pre condition, post condition, or class invariant), variable list, function name, function signature, and constraint. An example of an OCL query and its corresponding information generated by the OCL compiler is shown in Figure ??.

#### OCL Query:

```
Stack::pop():char
pre : self.top != 0
```

#### Query information and C++ generated code in file OCL.dat:

```
class = Stack
function = pop
action = pre
signature = bool pre_pop()
constraint =
  if ( self.top != 0)
    return 1;
  else {
    showResult("pre-condition violated, top == 0");
    return 0;
  }
```

Figure 4: OCL query. This figure illustrates an OCL query and the information generated by the OCL compiler.

### 3.2 The Back-end

The back-end is the part of our system where the debugging is actually accomplished. We use the technique of source code instrumentation at compile time; in this phase, OpenC++ plays a key roll. The OpenC++ MOP provides an efficient way of implementing customized language extensions that we use to implement a library called the CodeInstrumentor to instrument code into the debugged program. The CodeInstrumentor is a library that acts as an OpenC++ compiler plug-in. During the source-to-source translation, the CodeInstrumentor takes information in an output file

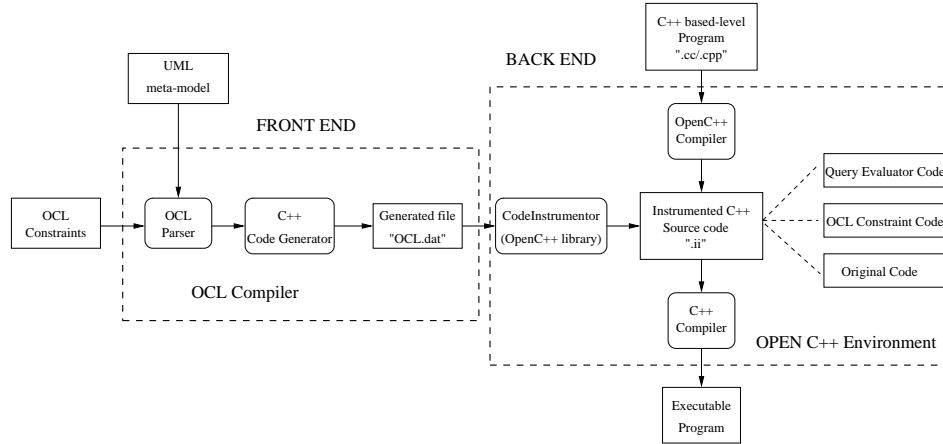


Figure 2: Overview of the C++ Query-Based Debugger.

from the compiled OCL queries of the front-end and inserts this information into the parse-tree of the pre-compiled base-level program where appropriate depending upon the constraints: pre-condition, post-condition, or class invariant. The parse-tree are Ptree meta-objects generated by the OpenC++ compiler from the base-level program during the pre-compilation stage. After the instrumentation is done, the instrumented program that consists of the original code, the constraint code, and/or the query evaluator, is again compiled by the ordinary C++ compiler. The final output is the executable program that include the query-based debugger. Whenever the constraint is met, the program in execution is terminated and report an error.

With only one line added before class definition of the class of interest, instrumentation of the class meta-objects can be done in the background while the original source code stays intact.

The declaration defines Stack as an instance of InstrumentorClass. So the Stack class becomes a class meta-object of the meta-class InstrumentorClass.

As aforementioned, when the OpenC++ compiler sees the annotation, it analyzes and decomposes the annotated class of the base-level C++ program, in this case the Stack class, into five aspects represented by meta-objects which are instances of meta-class Class, Ptree, Member, TypeInfo, and Environment. The compiler not only generates meta-objects for the Stack class but it does go through the inheritance hierarchy of the class Stack, if exists, and generates meta-objects for those classes as well. For other classes in the program that are not related with Stack and not declared as instances of the InstrumentorClass remain as is.

For a meta-class to be able to control the translation of its instance, it must be a subclass, direct or indirect, of the default meta-class Class. In this case, the meta-class InstrumentorClass indirectly inherits from the meta-class Class and overrides member functions of the Class. Since a class meta-object of the Stack is an instance of the meta-class Class, same as ptree, member, environment, and typeInfo meta-objects that are instances of meta-class Ptree, Member, Environment, and TypeInfo respectively, it is translated according to new member functions overridden in the InstrumentorClass.

The class diagram in Figure ?? shows relationship between the meta-class InstrumentorClass, its instance that is the Stack meta-object, the meta-class WrapperClass, the runtime support class QueryEvaluator, and basic meta-classes in OpenC++ environment. The CodeInstrumentor library shown in Figure ?? is the combination of InstrumentorClass and WrapperClass.

### 3.2.1 InstrumentorClass

The meta-class InstrumentorClass inherits from the meta-class WrapperClass<sup>1</sup> which again inherits from the default meta-class Class. The InstrumentorClass is the main class that controls the instrumentation of the program depending upon the OCL queries. It starts the process by constructing an instance of OCLInfo from the input file, OCL.dat. Then it checks the information to see what need to be performed. If the constraint is either pre-condition or post-condition, the translation of the class and its member function of interested is forwarded to the WrapperClass in order to generate a wrapper function. For post-condition, an extra step need to be performed in the InstrumentorClass since sometimes there is a reference to a value of the variable prior to invoking the operation, varpre. So the InstrumentorClass has to generates a new data member called varAtPre to keep track of the value of the variable before it is updated by the operation.

### 3.2.2 WrapperClass

The WrapperClass generates wrapper functions that perform as interceptor of calls to the original "wrapped" functions. When pre-condition or post-condition is specified in the OCL constraints, the control is transferred from the InstrumentorClass to the WrapperClass.

To generate the wrapper functions, the WrapperClass changes the name of the original function to org\_function. Then it generates a new wrapper function with the original function name The function body of this new member includes constraint from the query and also a call to the original function under the new name, org\_function. The location of the call to the original function can be before or after the

<sup>1</sup>This is a modified version of WrapperClass from [?].

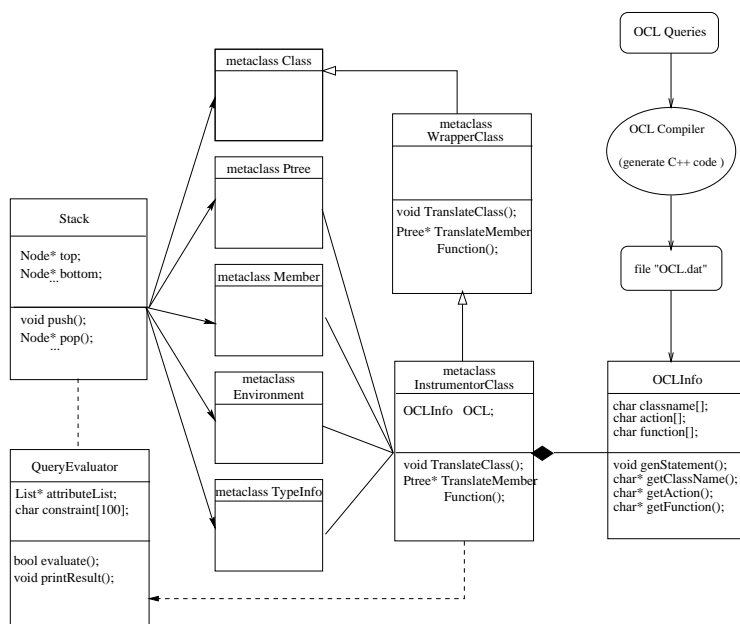


Figure 5: Class diagram representing the hierarchy of the meta-objects.

constraints depending upon the action of the constraint if it is pre-condition or post-condition.

### 3.2.3 QueryEvaluator

The QueryEvaluator acts as a runtime support code that evaluate the invariant of the class. It is generated on-the-fly by the InstrumentorClass during the source-to-source translation of the program when the constraint is on class invariant.

## 4. CONCLUDING REMARKS

We have described an approach for using the object constraint language, OCL, to formulate queries to facilitate debugging of C++ programs. Our use of OCL eliminates the problem of side-effect for queries formulated using the language of the underlying implementation. In addition, our approach of using OCL, together with the Unified Modeling Language, UML, enables the use of these queries during the design stage of software development. The queries formulated at the design stage can facilitate enforcement of design by contract as well as other design constraints. Furthermore, the queries formulated early in the design process can be reused during later design stages, and the queries can be reused during the debugging of the actual software. Our work is ongoing as we continue with our implementation using the OpenC++ meta-object protocol.