Doc no: N2216=07-0076
Date: 2007-03-11
Reply-To: Bjarne Stroustrup
bs@cs.tamu.edu

Report on language support for Multi-Methods and Open-Methods for C++

Peter Pirkelbauer
Texas A&M University
peter.pirkelbauer@tamu.edu

Yuriy Solodkyy
Texas A&M University
yuriys@cs.tamu.edu

Bjarne Stroustrup
Texas A&M University
bs@cs.tamu.edu

Abstract

Multiple dispatch – the selection of a function to be invoked based on the dynamic type of two or more arguments – is a solution to several classical problems in object-oriented programming. We present the rationale, design, and implementation of a language feature, called open multi-methods, for C++. Open multi-methods support both repeated and virtual inheritance and our call resolution rules generalize both virtual function dispatch and overload resolution semantics. After using all information from argument types, these rules can resolve further ambiguities by using covariant return types. We describe a model implementation and compare its performance and space requirements to existing open multi-method extensions and workaround techniques for C++. Compared to these techniques, our approach is simpler to use, catches more user mistakes (such as ambiguities), performs significantly better, and requires less memory. For example, our implementation of a multi-method call is constant-time and more than twice as fast as double dispatch - only 4% slower than a C++ virtual function call. Finally, we provide a sketch of a design for open multi-methods in the presence of dynamic loading and linking of libraries.

Keywords multi-methods, open-methods, multiple dispatch, object-oriented programming, generic programming, C++

1. Introduction

This technical report presents work in progress prompted by real-world problems, academic research, and discussions in the C++ standards committee (SC22/WG21). In particular, N1529 [28] is a specific proposal for adding a form of multimethods to the upcoming revision of the ISO C++ standard, C++0x. The aim of this TR is to provide a thorough (if still incomplete) discussion of the design alternatives, present a current best effort design, and present performance data from the current implementation demonstrating significant advantages over current workarounds.

Runtime polymorphism is a fundamental concept of object-oriented programming (OOP), typically achieved by late binding of method invocations. "Method" is a common term for a function chosen through runtime polymorphic dispatch. Most OOP languages (e.g.: C++ [31], Eiffel [24], Java [3], Simula [6], and Smalltalk [18]) use only a single parameter at runtime to determine the method to be invoked ("single dispatch"). This is a well-known problem for operations where the choice of a method depends on the types of two or more arguments ("multiple dispatch"), such as an intersect() function. A well-studied subset of this problem is the binary method problem [7]. Another problem is that dynamically dispatched functions have to be declared within class definitions. This often requires more foresight than class designers possess, complicating maintenance and limiting the extensibility of libraries.

Workarounds for both of these problems exist for single-dispatch languages. In particular, the visitor pattern (double dispatch) [16] circumvents the first problem without compromising type safety. Using the visitor pattern, the class-designer provides an accept method in each class and defines the interface of the visitor. This interface definition, however, limits the ability to introduce new subclasses and hence curtails program extensibility [11]. In [33] Visser presents a possible solution to the extensibility problem in the context of visitor combinators, which make use of RTTI.

Providing dynamic dispatch for multiple arguments lifts these restrictions. If declared within classes, such functions are often referred to as "multi-methods". If declared independently of the type on which they dispatch, such functions are often referred to as open class extensions, accessory functions [35], arbitrary multi-methods [26], or "open-methods". Languages supporting multiple dispatch include CLOS [29], MultiJava [11, 25], Dylan [27], and Cecil [9]). We implemented and measured both multi-methods and open-methods. Since open-methods address a larger class of design problems than multi-methods, our discussion concentrates on open-methods.

Generalizing from single dispatch to open-methods raises the question how to resolve function invocations in cases where no overrider provides an exact type match for the runtime-types of the arguments. Symmetric dispatch treats each argument alike but is subject to ambiguity conflicts. Asymmetric dispatch resolves conflicts by ordering the argument based on some criteria – typically, an argument list is considered left-to-right). Asymmetric dispatch semantics is simple and ambiguity free (if not necessarily unsurprising to the programmer), but it is not without criticism [8]. In addition, asymmetric dispatch differs radically from C++'s symmetric function overload resolution rules.

We derive our design goals for the open-method extension from the C++ design principles outlined in [30]:

- A language extension should address several specific problems.
- A new mechanism should not impose costs on code that does not use it. In this case, open-methods should neither prevent separate compilation of translation units nor increase the cost of ordinary virtual function calls.
- Code using a new language feature should benefit compared to code that uses workaround techniques. In this case, open-

ISO/IEC JTC1/SC22/WG21
Doc no: N2216=07-0076
Date: 2007-03-11
Reply-To: Bjarne Stroustrup (bs@cs.tamu.edu).

methods should be more convenient to use than all workarounds (e.g. the visitor pattern) as well as outperforming them in both time and space.

- Semantics introduced by a new mechanism should fit well with existing features. In particular, open-methods should be unsurprising when compared to virtual and overloaded functions.
- The mechanism should be general and useful for a wide variety of systems. In particular, exception handling is not currently considered suitable for hard real-time system (e.g. [23]) so throwing exceptions to indicate an ambiguity conflict is best avoided.

Section 2 presents application domains for both open-methods and multi-methods. Section 3 describes our function call and ambiguity resolution mechanisms. Section 4 shows the necessary modifications to the C++ compiler and linker strategy as well as extensions of the IA-64 object model [12] based on our model implementation. Section 6 discusses problems related to dynamic loading and linking of libraries. Section 7 gives an overview of research in the area of multi-methods for C++ and other languages. Section 8 compares the performance of our approach to other methods that add support for multi-methods to C++. Section 9 summarizes our contributions and sketches remaining open problems.

2. Application Domains

The question whether open-methods address a sufficient range of problems to be a worthwhile language extension is a popular question. We think they do, but do not consider the problem one that can in general be settled objectively, so we just present examples that would benefit significantly. We consider these examples characteristic for larger classes of problems.

2.1 Shape Intersection

An intersect operation is a classical example of multi-methods usage [30]. For a hierarchy of shapes, intersect() decides if two shapes intersect. Handling all different combinations of shapes (including those added later by library users) can be quite a challenge. Worse, a programmer needs specific knowledge of a pair of shapes to use the most specific and efficient algorithm.

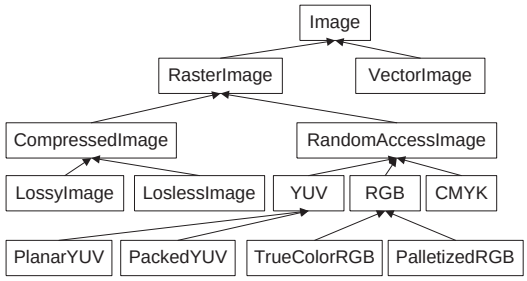
Using the multi-method syntax from [30], with **virtual** indicating runtime dispatch, we can write:

```
bool intersect(virtual Shape&, virtual Shape&); // open−method
bool intersect(virtual Rectangle&, virtual Circle&); // overrider
```

We note that for some shapes, such as rectangles and lines, the cost of double dispatch can exceed the cost of the intersect algorithm itself.

2.2 Data Format Conversion

Consider an application, such as an image processor or a web browser that deals with many image formats and must frequently convert between them. Generic handling of formats by converting them to and from a common representation in general gives unacceptable performance, degradation in image quality, loss of information, etc. For example, conversions between an RGB and a YUV format are computation intensive. However, conversions between different RGB formats and between different YUV formats can be done simply and efficiently. Here is the top of a realistic image format hierarchy:



A host of concrete image formats such as RGB24, JPEG, and planar YUY2 will be represented by further derivations. The optimal conversion algorithm must be chosen based on a source-target pair of formats [20] [36]. That is, we again need a lookup based on two runtime types from a large and extensible hierarchy.

2.3 Binary operations

Most forms of computation involve many types and binary operations. Matrix algebra is an obvious example. For example:

```
void computation(const Matrix& a, const Matrix& b)
{
  Matrix tmp = a+b; // binary operation
  // ...
}
```

Often, operations are selected based on static types, rather than relying on a base class as in the example. The reason for that is to improve performance, to eliminate the complexity of double dispatch, and to gain the benefits of predictable ambiguity resolution. Open-methods address those concerns.

The implementation of a scripting language would be an application where the solution to the binary operation problem would be performance sensitive.

3. Definition of open-methods

Open-methods are dynamically dispatched functions, where the callee depends on the dynamic type of one or more arguments. ISO C++ supports compile-time (static) function overloading and runtime (dynamic) dispatch on a single argument. The two mechanisms are orthogonal and complementary. We define open-methods to generalize both, so our language extension must unify their semantics. Our dynamic call resolution mechanism is modeled after the overload resolution rules of C++. The ideal is to give the same result as static resolution would have given had we known all types at compile time. To achieve this, we treat the set of overriders as a viable set of functions and choose the single most specific method for the actual combination of types.

We derive our terminology from virtual functions: a function declared virtual in a base class (super class) can be overridden in a derived class (sub class):

- an open-method is a non-member function with one or more parameters declared virtual
- an overrider is an open-method that refines another open-method according to the rules defined in §3.1
- an open-method that does not override another open-method is called a base-method.

For example:

```
struct A { virtual ~A(); };
struct B : A {};

void print(virtual A&, virtual A&); // (1)
```

```
void print(virtual B&, virtual A&); // (2)
void print(virtual B&, virtual B&); // (3)
```

Here, both (2) and (3) are overriders of (1), allowing us to resolve calls involving every combination of A's and B's. For example, a call print(a,b) will involve a conversion of the B to an A and invoke (1). This is exactly what both static overload resolution and double dispatch would have done.

To introduce the role of multiple inheritance, we can add to that example:

```
struct X { virtual ~X(); };
struct Y : X, A {};

void print(virtual X&, virtual X&); // (4)
void print(virtual Y&, virtual Y&); // (5)
```

Here (4) defines a new open-method print on the class hierarchy rooted in X. Y inherits from both A and X, and since both print open-methods have the same signature, – (5) is an overrider for both (4) and (1).

3.1 Overriding

DEFINITION 1. *An open-method is considered an overrider (*or*) for an open-method (*om*) in the same translation unit if it has:*

- *the same name*
- *the same number of parameters*
- *possibly covariant virtual parameter types*
- *invariant non-virtual parameter types*

A base-method must be declared before any of its overriders. This restriction parallels other C++ rules and greatly simplifies compilation. As shown in the previous example, an overrider can be associated with more than one base-method.

For every overrider and base-method pair, the compiler checks, if the exception specifications comply with the rules used for virtual functions and if the overriders comply with covariant return type semantics.

DEFINITION 2. *An open-method that is not an overrider and an overrider that introduces a covariant return type are considered a* base-method *for a translation unit.*

DEFINITION 3. *A* Dispatch table (DT) *maps the type-tuple of the base-method's virtual parameters to actual overriders that will be called for that type-tuple.*

Millstein and Chambers show in [26] that open-methods cannot be modularly type checked if the language supports multiple implementation inheritance. Therefore, we split our call resolution mechanism into three distinct stages:

- Overload resolution
- Ambiguity resolution
- Run-time dispatch

The goal of overload resolution is to find at compile time a unique base-method, through which the call can be dispatched. We note, that this base-method will not be used for the actual dispatch at run-time, but rather to determine a dispatch table through which the call will be made, the necessary casts of the arguments and the expected return type. The actual overrider to handle the call will only be determined at run-time.

The C++ overload resolution rules [21] are unchanged: the viable set includes both open-methods and regular functions and treats open-methods like any other free-standing functions. Dynamic dispatch is used only if an open-method is the best match. We relax this rule slightly: if a set of best matches consists of open-methods only and the intersection of their base-methods has a single element - overload resolution does not report an ambiguity. We demonstrate with an example:

```
struct X;
struct Y;
struct Z;

void foo(virtual X&, virtual Y&); // (1)
void foo(virtual Y&, virtual Y&); // (2)
void foo(virtual Y&, virtual Z&); // (3)

struct XY : X, Y {}
struct YZ : Y, Z {}

void foo(virtual XY&, virtual Y&); // (4)
void foo(virtual Y&, virtual YZ&); // (5)
```

Open-methods 1,2 and 3 are three independent base-methods defined on different class hierarchies. Because XY and YZ are parts of several hierarchies, overriders 4 and 5 refine several base-methods. In particular 4 is an overrider for 1 and 2 and 5 is an overrider for 2 and 3.

A call foo(xy,yz); with arguments of types XY and YZ respectively is now ambiguous according to the standard overload resolution rules as both 4 and 5 are equally good matches. Our relaxed rule, however, does not reject this call as ambiguous at compile time, because these overriders have a unique base-method through which the call can be dispatched – 2.

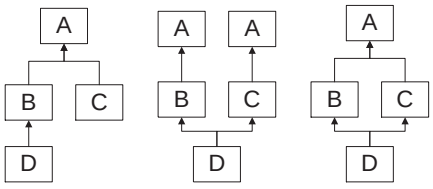
At link time, when all the overriders have been seen, we check the overriders for return type consistency, perform ambiguity resolution and build the dispatch tables. We describe this stage more in 3.2.

Run-time dispatch simply looks up the entry in the dispatch table that corresponds to the dynamic types of the arguments and dispatches to that function.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [19]. Further, the use of open-methods (as opposed to ordinary virtual functions and multi-methods) can be seen as adding a runtime dimension to generic programming [4].

3.2 Ambiguity resolution

C++ supports single-, repeated-, and virtual inheritance:



Note that to distinguish repeated and virtual inheritance, this diagram represents sub-object relationships, not just sub-class relationships. We must handle all ambiguities that can arise in all these cases. By "handle" we mean resolve or detect as errors.

Our ideal for resolving open-method calls is the union of the ideals for virtual functions and overloading:

- virtual functions: the same function is called independently of which sub-type in an inheritance hierarchy is used in the call.
- overloading: a call is considered unambiguous if (and only if) every parameter is at least as good a match for the actual argument as the equivalent parameter of every other candidate function and that it has at least one parameter that is a better match than the equivalent parameter of every other candidate function.

This implies that a call of a single-argument open-method is resolved equivalently to a virtual function call. The rules described below closely approximate this ideal. As mentioned, the static resolution is done exactly according to the usual C++ rules. The dynamic resolution is presented as the algorithm for generating dispatch tables in §3.4. Before looking at that algorithm, we present some key motivating examples.

3.2.1 Single Inheritance

In object models supporting single inheritance (§3.2) ambiguities can only occur with open-methods taking at least two virtual parameters. Ambiguities in this case have to be resolved by introducing a new overrider. The resolution of an open-method with one argument is identical to that of a virtual function. Thus, open-methods provide an unsurprising mechanism for expressing non-intrusive ("external") polymorphism.

3.2.2 Repeated Inheritance

Consider the repeated inheritance case (§3.2) together with this set of open-methods visible at a call site to foo(d1,d2):

```
void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
```

Every foo() is a match, but is one a best match? No, the usual overload resolution rules reject that call, and the compiler reports the ambiguity immediately. The result of overload resolution determines the base-method through which the call will be dispatched. The choice of this method will affect casting of argument types at the call site and determine the expected return type (in the presence of covariant return). To resolve that ambiguity, a user can either add an overrider foo(D&,D&) visible at the call site or explicitly cast arguments to either the B or C sub-object.

When the above ambiguity is resolved by casting, a question still remains on how the pre-linker should resolve a call with two arguments of type D? We know at runtime (by looking into the virtual function table's open-method table (see §4) which "branch" of a D object (either B or C) is on. Thus, we can fill our dispatch table appropriately; that is, for each combination of types there is a unique "best match" according to the usual C++ rules:

	A	B	C	D/B	D/C
A	AA	AA	AA	AA	AA
B	AA	BB	BC	BB	BC
C	AA	CB	CC	CB	CC
D/B	AA	BB	BC	BB	BC
D/C	AA	CB	CC	CB	CC

This depicts the dispatch table for the repeated-inheritance hierarchy in §3.2 and the set of overriders above. Since the base method is foo(A&,A&) and A occurs twice in D, each dimension has two entries for D: D/B meaning "D along the B branch". This resolution exactly matches our ideals.

3.2.3 Virtual Inheritance

Consider the virtual inheritance class hierarchy from §3.2 together with the set of open-methods from §3.2.2: In contrast to repeated inheritance, a D has only one A part, shared by B, C, and D. This causes a problem for calls requiring conversions, such as foo(b,d); is that D to be considered a B or a C? There is not enough information to resolve such a call. Note that the problem can arise is such a way that we cannot catch it at compile time:

```
C& rc = d;
foo(b,rc);
B& rb = d;
foo(b,rb);
```

Using static type information to resolve either call would violate the fundamental rule for virtual function calls: thus, use runtime type information to ensure that the same overrider is called from every point of a class hierarchy. At runtime, the dispatch mechanism will (only) know that we are calling foo with a B and a D. It is not known whether (or when) to consider that D a B or a C. Based on this reasoning (embodied in the algorithm in §3.4) we must generate this dispatch table:

	A	B	C	D/A
A	AA	AA	AA	AA
B	AA	BB	BC	??
C	AA	CB	CC	??
D/A	AA	??	??	??

We cannot detect the ambiguities marked with ?? at compile time, but we can catch them at link time when the full set of overriders are known.

3.3 Covariant return types

Covariant return types are a useful element of C++. If anything they appear to be more useful for operations with multiple arguments than for single argument functions. For example, consider a class Symmetric derived from Matrix:

```
Matrix& operator+(Martix&, Matrix&);
Symmetric& operator+(Symmetric&, Symmertic&);
```

It follows that we must generalize the covariant return rules for open-methods. Doing so turned out to be unexpectedly useful because covariant return types help resolve ambiguities.

In single dispatch, covariance of a return type implies covariance of the receiver object. Consequently, covariance of return types for open-methods imply an overrider (or) - base-method (bm) relationship between two open-methods. Liskov's substitution principle [22] guarantees that any call type-checked based on bm can use or's covariant result without compromising type safety.

This can be used to eliminate what would otherwise have been ambiguities. Consider the class hierarchies $A \leftarrow B \leftarrow C$ and $R1 \leftarrow R2 \leftarrow R3 \leftarrow R4$ and this set of open-methods:

```
R1* foo(virtual A&, virtual A&);
R2* foo(virtual A&, virtual B&);
R3* foo(virtual B&, virtual A&);
R4* foo(virtual B&, virtual C&);
```

A call foo(b,b) appears to be ambiguous and the rules outlined so far would indeed make it an error. However, choosing R2* foo(A&,B&) would throw away information compared to using R3* foo(B&,A&): An R3 can be used wherever an R2 can, but R2 cannot be used wherever an R3 can. So we prefer a function with a more derived return type and for this example get the following dispatch table:

	A	B	C
A	AA	AB	AB
B	BA	BA	BC
C	BA	BA	BC

At first glance, this may look useful, but ad hoc. However, an open-method with a return type that differs from its base method becomes a new base method and requires its own dispatch table (or equivalent implementation technique). The fundamental reason is the need to adjust the return type in calls. Obviously, the resolutions

for this new base method must be consistent with the resolution for its base method (or we violate the fundamental rule for virtual functions). However, since R2∗ foo(A&,B&) will not be part of R3∗ foo(B&,A&)'s table, the only consistent resolution is the one we chose.

If the return types of two overriders are siblings, then there is an ambiguity in the type-tuple that is a meet of the parameter-type tuples. Consider for example that $R3$ derives directly from $R1$ instead of $R2$, then none of the existing overriders can be used for (B,B) tuple as its return type on one hand has to be a subtype of $R2$ and on the other a subtype of $R3$. To resolve this ambiguity, the user will have to explicitly provide an overrider for (B,B), whose return type must derive from both $R2$ and $R3$.

Using the covariant return type for ambiguity resolution also allows the programmer to specify preference of one overrider over another when asymmetric dispatch semantics is desired.

To conclude: covariant return types do not only improve static type information, but also enhance our ambiguity resolution mechanism. We are unaware of any other multi-method proposal using a similar technique.

3.4 Algorithm for dispatch table generation

Let us assume we have a multi-method $rf(h_1, h_2, ..., h_k)$ with k virtual arguments. Class h_i is a base of hierarchy of the i^{th} argument. $H_i = \{c : c <: h_i\}$ is a set of all classes from the hierarchy rooted at h_i. $X = H_1 \times H_2 \times \cdots \times H_k$ is the set of all possible argument type-tuples of f. Set $Y = \{(y_1, y_2, \cdots, y_k)\} \subseteq X$ is the set of argument type-tuples, on which the user defined overriders f_j for f. The set $O_f = \{f_0, \cdots, f_{m-1}\}$ is the set of those overriders ($f_0 \equiv f$). A mapping $F : Y \leftrightarrow O_f$ is a bijection between type-tuples on which overriders are defined and the overriders themselves.

Because different derivation paths may get different entries in the dispatch table, we assume that x_i in the type-tuple $x = (x_1, \cdots, x_k)$ identifies not only the concrete type, but also a particular derivation path for it (see [34] for formal definitions). Under this assumption, we define $B(x_i)$ to be a direct ancestor (base-class) of x_i in the derivation path represented by x_i. For example, for the repeated inheritance hierarchy from §3.2, $B(D/B) = B, B(D/C) = C, B(C) = A$, while for the virtual inheritance hierarchy $B(D/A) = A, B(B) = A, B(C) = A$.

For the sake of convenience we define:

$$B_i(x) \equiv (x_1, \cdots, B(x_i), \cdots, x_k).$$

With it we extend the definition of B to type-tuples as follows:

$$B(x) \equiv \{B_1(x), B_2(x), \cdots, B_k(x)\}.$$

$P(X, <) : (x_1, ..., x_k) <_P (y_1, ..., y_k) \Leftrightarrow \forall i : x_i <: y_i \wedge \exists j : y_j \not<: x_j$ defines a partial ordering that models ordering of viable functions for overload resolution as defined in [21]. $max\_set(S) = \{x \in S \subseteq X : \nexists y \in S : x < y\}$ is a set of maximal elements of S with respect to the partial ordering P.

Dispatch table DT is a mapping $DT : X \rightarrow O_f$ that maps various combinations of argument types to the overriders used to handle that combination.

For any combination of argument types $x \in X$, we recursively define entries of the dispatch table DT as following:

$$DT[x] = \begin{cases} F(x), x \in Y \\ DT[max\_set(B(x))], |max\_set(B(x))| = 1 \\ Ambiguity, otherwise \end{cases}$$

The above recursion exhibits optimal substructure and has overlapping sub-problems, which lets us use dynamic programming [13] to create an efficient algorithm for generation of dispatch table, shown in Algorithm 1.

Algorithm 1 Dispatch Table Generation

```
S ← topological_sort(X)
for all x ∈ S do
  if x ∈ Y then
    DT[x] ← F(x)
  else
    max_set = {B_1(x)}
    for i ← 2, k do
      dominated ← false
      for all e ∈ max_set do
        if F^-1(DT[e]) <_P F^-1(DT[B_i(x)]) then
          max_set ← max_set − {e}
        else if F^-1(DT[B_i(x)]) <_P F^-1(DT[e]) then
          dominated ← true
          break
      if not dominated then
        max_set ← max_set ∪ {F^-1(DT[B_i(x)])}
    if |max_set| = 1 then
      DT[x] ← F(max_set)
    else
      Report ambiguity for x
```

To analyze its performance, we first note that comparison of two type-tuples from X can be done in time $O(k)$. If $n = max(|H_i|, i = 1, k)$ and $r = max(r_i, i = 1, k)$ (where r_i is a maximum number of times h_i is used as non-virtual base class in any class of hierarchy H_i) then $|X| <= (n * r)^k$ and the amount of edges for topological sort is less then $k * (n * r)^k$. Therefore the complexity of topologically sorting X is $O(k * n^k)$. The second loop has complexity $O(k^2 * n^k)$ so the overall complexity is $O(n^k)$ since k is a constant defining the amount of virtual arguments. This means that the algorithm is linear in the size of the dispatch table.

3.5 Alternative dispatch semantics

Our open-method semantics strictly corresponds to virtual member function semantics in ISO C++ but does not entirely reflect overload resolution semantics. The reason is that less information is available for compile-time resolution that for link-time or run-time resolution. For example, consider the repeated inheritance class hierarchy from §3.2 with a virtual function added:

```
struct A { virtual void foo(); };
struct B : A {};
struct C : A { virtual void foo(); };
struct D : B, C {};

void bar(A&); // conventional overloading
void bar(C&);

void foobar(virtual A&); // open−method
void foobar(virtual C&); // open−method


D d;
B& db = d; // B part of D
C& dc = d; // C part of D

// (run−time) Virtual Member Function Semantics:
b.foo(); // calls A::foo
c.foo(); // calls C::foo
d.foo(); // error: ambiguous

// (compile−time) Overload Resolution Semantics:
bar(db); // calls bar(A&)
bar(dc); // calls bar(C&)
```

```
bar(d); // calls bar(C&) (why not ambiguous?)

// (runtime−time) open−method Semantics:
foobar(db); // calls foobar(A&)
foobar(dc); // calls foobar(C&)
foobar(d); // error: ambiguous
```

The difference between the ordinary virtual function (foo) calls and the ordinary overloaded resolution for (bar) is odd and depends on pretty obscure rules that may be more historical than fundamental. The open-method (foobar calls follows the virtual function resolution.

Further differences emerge in cases where a different resolution become possible in cases where additional information from other translation units may become available to resolve open-methods (see §5 and §6.2). This parallels decisions for related parts of the language. For example, the resolution of **static_cast** and **dynamic_cast** can differ even given identical arguments: **dynamic_cast** can use more information than **static_cast**.

4. Implementation

We implemented open-methods as described here by modifying the EDG compiler front-end [14].

4.1 Changes to Compiler & Linker

Our mechanism extends ideas presented in [15, 35] as to compiler and linker model. We adopted the multi-method syntax proposed in [30], which in turn was inspired by an early idea by Doug Lea. One or more parameters of a non-static freestanding function can be specified to be **virtual**. A virtual argument must be a reference or pointer to a polymorphic class (that is, a class containing at least one virtual function). For example:

```
struct A { virtual ~A(); };

void print(virtual A&); // ok
void print(int, virtual A&); // ok

void dump(virtual Shape); // compiler error
void dump(virtual int); // compiler error
```

Open-methods are generic free-standing functions, which do not have the access privileges of member functions. If an open-method needs access to non-public members of a class, that class must declare it to be a friend. An open-method must be defined; that is, there are no abstract (pure virtual) open-methods. For example, we must define the intersect of shapes:

```
bool intersect(virtual Shape&, virtual Shape&) { }
```

We could allow multi-methods for abstract classes to be abstract. The obvious implementation would be to call an error function of some sort (like for a virtual function). Similarly, the obvious syntax would be the =0 (like for a virtual function).

For each translation unit, the compiler generates an *open-method description* (OMD) file that stores the data needed to generate the runtime data-structure discussed in §4.2. Essentially, this includes the names of all classes, their inheritance relationships, as well as the kind of inheritance. Open-methods are represented by name, return-type, and their parameter-list. Finally, the OMD-file also contains definitions of all user-defined types that appear in signatures of open-methods (both as virtual and regular parameters). These definitions are necessary to regenerate prototype declarations for the open-methods, which pass data through by value.

Information collected in OMD-files is assembled together by the pre-linker, which is invoked last in the compilation chain, before the object code of all translation units is linked together. The pre-linker will synthesize each base-method with its overriders into a dispatch table, issue link-errors for ambiguities, determine the indices necessary to access the open-method and dispatch-table, as well as define and interlink the om-tables of each sub-object type as described in §4.2. When the call of an overrider requires adjustments of the this-pointers (as is sometimes needed in multiple inheritance hierarchies), the pre-linker creates thunks and makes the dispatch table entries refer to them instead. During dispatch table synthesis, the linker will report errors for all argument-combinations, which do not have a unique best overrider. The output of the pre-linking stage is a C-source file containing the missing definitions. If the linker generates a library, the pre-linker also puts out a merged OMD-file.

4.2 Changes to Object Model

We augment the IA-64 C++ object model [12] by four elements to support constant time dispatching of open-methods. First, for each base-method there will be a dispatch table containing the function addresses. Second, the v-table of each sub-object contains an additional pointer to the *open-method table (om-table)*. Finally, the indices used for the open-method-table offsets are stored as global variables.

Figure 1 shows the layout of objects, v-tables, om-tables and dispatch-tables for repeated (left) and virtual (right) inheritance. Our extensions to the object-model are shown with grey background. From left to right the elements in each diagram represent the object, v-table, om-table, and dispatch table(s) for the class hierarchy in §3.2. From top to the bottom, the objects are of type A, B, C, and D respectively.

An open-method can be declared after the declarations of the classes used in its virtual parameters. Therefore, the compiler cannot reserve v-table entries to store the data related to open-method dispatch immediately in a class's virtual function table. Hence, we always extend every v-table by one pointer referencing the om-table, which can be laid down later by the pre-linker.

The om-table reserves one position for each virtual parameter of each base-method, where objects of this type can be passed as arguments. This position stores an index into corresponding dimension of the dispatch table. Since the size of the om-tables is not known at compile-time, our technique relies on a literal for each open-method and virtual parameter position (called foo_1st, foo_2nd in Figure 1) that determines the offset within the om-tables.

Note that Figure 1 depicts our actual implementation, where entries for first argument positions already resolves one dimension of the table lookup. Entries for all other argument positions store the byte offset within the table.

In the presence of multiple-inheritance, a this-pointer shift might be required to pass the object correctly. In this case, we replace the address of the overrider by an address of a thunk that takes care of correctly adjusting the this-pointer. As described in §3.2.2 in case of repeated inheritance different bases can show different dispatch behavior depending on the sub-object to which the this-pointer refers. As a result, different bases may point to different om-tables. In case of virtual inheritance, the open-method dispatch entries are only stored through the types mentioned in the base-method. Hence, in the virtual inheritance case, all open-method calls are dispatched through the virtual base type.

4.3 Alternative Approaches

We considered a few other design alternatives to explore their trade-offs in extensibility and performance.

4.3.1 Multi-Methods

Multi-methods differ from open-methods in that the base-method has to be declared in the class definition of its virtual parameters.

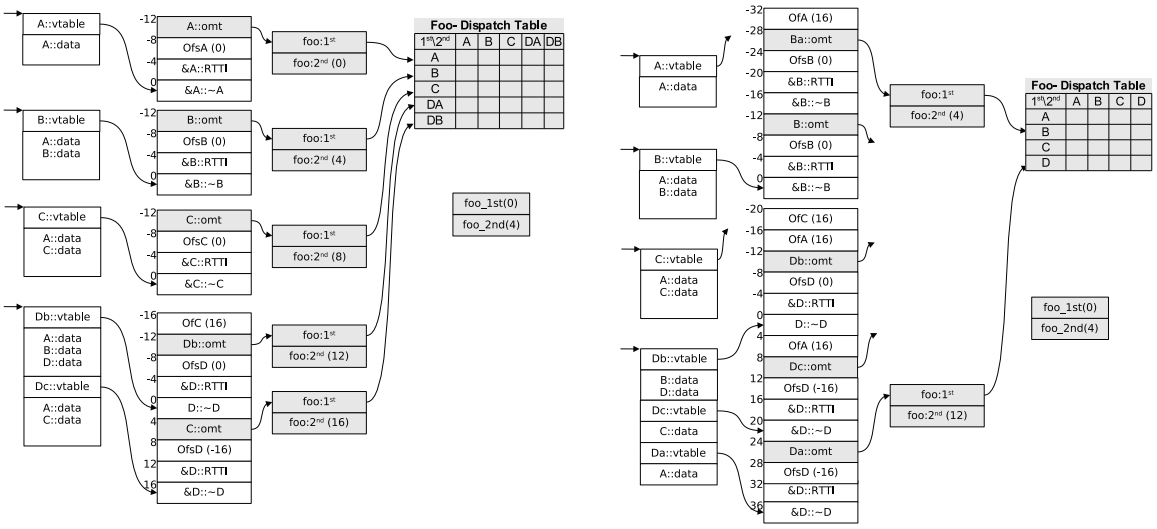


Figure 1. Object Model for repeated (left) and virtual (right) inheritance

This allows the offset within the v-table be known at compile time, which saves two indirections per argument of a function call (one for the om-table, and one to read the index within the om-table). For a call with k virtual arguments, open-methods need $4k + 1$, while multi-methods need only $2k + 1$ memory references to dispatch a call. The downside of multi-methods is that existing classes cannot easily be extended with dynamically dispatched functions. Consider:

```
class Matrix
{ // multi−method declaration
  friend Matrix& operator+( virtual const Matrix& lhs,
                            virtual const Matrix& rhs );
  // ...
  virtual Matrix& operator∗(virtual const Matrix&);
};
```

We implemented only the non-member version of multi-methods. The member version can be implemented with exactly the same techniques. However, in many cases it is harder to write code that uses the member version because an overrider must be a member of (only) one class – and the main rationale for multi-methods is to elegantly deal with combinations of classes. Even the non-member (friend) version is hard to use. By requiring a declaration to be present in a class, we limit the polymorphic operations those that the class designer thought of. That requires too much foresight of the class designer or leads to unstable classes (classes that keep having multi-methods added).

Such problems are well-known in languages relying on member functions. Open-methods provide an abstraction mechanism that solves such problems by separating operations from classes.

4.3.2 Chinese Remainders

In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [17]. The proposed scheme circumvents the necessity for open-method tables by moving all the necessary information from the class to the dispatch table.

Suppose that for every multi-method f there is a function $I_f : T \times N \rightarrow N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns index of type t in the n^{th} dimension of the f's dispatch table. If such function is reasonably fast (preferably constant time) and its range is small (preferably from 0 to maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging rows and columns accordingly to the indices returned by I_f. As in [17] we use the Chinese reminder theorem [13] to generate function I_f.

Despite its elegance, this approach is rather theoretical because it is hard to use for large class hierarchies. The reason is that we need to assign different prime numbers to classes and perform computations on numbers that are bound by the product of these primes. Such product can fit into 32-bit integers for only the 9 first primes and into 64-bit integers for only the 15 first primes. Table compression techniques [2] or use of minimal perfect hash functions [13] instead, can help overcome the problem.

5. Discussion

Ambiguities can arise at various stages in the build process. We describe here alternatives in their handling as well as various issues that may arise during development.

During compilation of a translation unit the compiler may take into consideration all the overriders seen in that translation unit as well as all the classes visible there, with which an open-method can potentially be called. The compiler then may report an error, when the dispatch table cannot be built unambiguosly using the overriders seen. Alternatively it may pospone ambiguity resolution to the next stage, where more overriders can be seen. We opt for postponing ambiguity resolution to the linking stage to allow other translation units to contribute in specializing. Resolving ambiguity at compile time seems inappropriate, because that same pair of types may be resolved differently in another translation unit.

Ambiguities detected at link time are due to inability to pick a unique best match for a particular combination of argument types. Again, we can report an error here, forcing the user to explicitly resolve it by providing an unambiguous overrider for that argument tuple. Alternatively we can postpone it till load time, hoping that some dynamic module will resolve the ambiguity. Further on we show a case in which we may also want to resolve ambiguity at link time due to user's physical inability to resolve it.

Loading several dynamically linked modules at run-time may again result in ambiguities even when each of the modules did not have any ambiguities at link time (see §6). This usually happens

because of the new classes introduced by modules as well as because of overriders that clash with overriders from other modules. Taking this into account, we disliked termination of an application (error) as user will not be able to intercept such scenario. We did not like throwing exceptions on ambiguities to allow for embedded systems. Besides, introduction of a single overrider can easily invalidate most of the dispatch table, which may be counterintuitive to the user.

We summarize the possibilities (marked with ○) and the choices we've made (marked with ●) in the following table:

Stage	Error	Postpone	Resolve	Throw
Compile time	○	●		
Link time	●	○	●	
Load time	○		●	○

5.1 "Hidden" classes

To resolve ambiguities at link time, the programmer needs access to the class definitions for which there is no unique best overrider. We call classes not available at link time "hidden" classes. Hidden classes can occur for two reasons. First, all local classes are considered to be hidden, because their name is local to the function scope in which they were defined. Second a programmer cannot access classes that are defined within a library or an implementation file and for which no definition is available in header files. We demonstrate with an example:

```
// Available common header
class A {};

void foo(virtual A&, virtual A&); // base method

class B : A {};
class C : A {};

// B1.cpp
class B1 : B {}; // definition of B1 is not available to others
void foo(virtual B1&, virtual A&);

// C1.cpp
class C1 : C {}; // definition of C1 is not available to others
void foo(virtual A&, virtual C1&);
```

The linker will find an ambiguity when it decides which overrider to call for a pair (B1,C1); however the user might not be able to resolve it because definitions of B1 and C1 are not available to him.

Since in this case the programmer will not be able to write code resolving ambiguities, we suggest to treat these cases equal to ambiguity resolution for dynamically linked libraries from §6.2. Only if the linker can determine, that ambiguities are resolveable, it must report an error.

Following, we will briefly describe a possible mechanism that could sufficiently well distinguish available from hidden classes. Such a mechanism could for example record all ambiguities that occur within at compile time of a translation unit. Hence, these ambiguities would be constrained by the class hierarchies and overriders seen at compile time. At link time, all these recorded ambiguities need to be resolved, otherwise the linker would report an error. After checking all recorded ambiguities, the linker can use rules described in §6.2 to resolve remaining ambiguities. By definition, libraries have been seen by the prelinker, and are therefore ambiguity free.

5.1.1 Open-methods and namespaces

Virtual functions have a class scope and can only be overriden in the derived classes. Open-methods do not have such a scope by default, so the question arises when should an open-method be considered an overrider and when just a different open-method? Let's look at the following example:

```
namespace X
{
  class A {};
  void bar(virtual A&); // base method

  class B : A {};
  void bar(virtual B&); // (1)
}

namespace Z
{
  void bar(virtual B&); // (2)
}

namespace Y
{
  class D : X::A {};
  void bar(virtual D&); // (3)
}

class C : X::A {};
void bar(virtual C&); // (4)
```

One approach may be to require overrider be declared in the same namespace as its base-method (1). In such scenario, open-methods with the same name and compatible parameter types, defined in different namespaces would not be considered overriders. Among advantages of this approach is easiness of understanding and implementation. Unfortunately such semantics is not unifiable with regular virtual function calls, where derived classes can be declared in a different namespace. We note that using declaration could potentially be used to work around these limitations.

Second alternative would be to let overriders be declared in any namespace (1,2,3,4). It is easy to understand but defeats the purpose of namespaces that were introduced to better structure the code and avoid name-clashes among indpendently developed modules.

Third approach may consider an open-method to be an overrider, if its base-method is defined in the same scope or in the scope of their argument types and their base classes. In this scenario (1, 3, 4) would override; (2) would not. Among its pros is its resemblance to argument dependent lookup. It would also work for virtual functions. Its cons, however, is that it is not easily comprehendable.

6. Dynamic linking

Outside embedded systems, dynamically linked libraries are almost universally used with C++. Thus, a design for open-methods that does not allow for DLLs is largely theoretical. We do not currently have an implementation, but here we outline a design addressing the major issues for open-methods in a dynamically linked library. It guarantees that the most specialized overrider available at runtime that preserves type-safety of a call will be used to dispatch a call.

Dynamic modules, compiled independently, may have different sets of overriders defined at the time of compilation. Furthermore, there could be new classes added to a hierarchy in one of the modules and objects of those classes may be passed into code of other modules. This is not a problem for regular virtual functions as their v-tables are found in the module where the class was defined. In case of open-methods, the dispatch table generated within a particular module can be simply unaware of a class, defined somewhere else. That is, the dispatch table for an open-method may lack the rows and columns needed to handle the class. To deal with this we

can either update each module's dispatch table with new classes (at load time) or keep a shared global dispatch table (updated by the loader).

We first argue why the second option is not viable and why each module that can be dynamically loaded into the process should have its own dispatch table.

6.1 Consistency of covariant return types

Covariant return introduces subtleties when dynamic linking is used. Consider a two-class hierarchy $A \leftarrow B$ and another two-class hierarchy $R1 \leftarrow R2$. The base-method R1 foo(**virtual** A&, **virtual** A&) is defined in a header visible by two dynamically linked modules D_1 and D_2 that do not know anything about each other. Module D_1 introduces overrider R2 foo(A&, B&) and module D_2 introduces overrider R1 foo(B&, B&). Each of the dynamically linked modules perfectly type-checks and links with foo() resolved through the dispatch table (a superscript in a cell denotes type that is returned by appropriate overrider e.g. AB^2 denotes R2 foo(A&, B&)):

D_1	A	B
A	AA^1	AB^2
B	AA^1	AB^2

D_2	A	B
A	AA^1	AA^1
B	AA^1	BB^1

When both modules are loaded together we get the dilemma of how to resolve a call with both arguments of type B: on one side foo(B&,B&) from D_2 is more specialized, but on the other hand foo(A&,B&) from D_1 imposes additional requirement that return type of whatever is called for (B,B) should be a subtype of R2, which R1 is not. Keeping a unique shared dispatch table for all the modules will force us to choose between suboptimal and type unsafe alternatives. What's worth - is that there may not be a unique type-safe alternative.

Imagine for example that a module D_3 introduces overrider R3 foo(B&, A&) where $R1 \leftarrow R3$, so $R2$ and $R3$ are siblings. When D_1 and D_3 are loaded together, neither R2 foo(A&, B&) nor R3 foo(B&, A&) can be used to resolve a call with both arguments of type B - both alternatives are type unsafe for the other overrider.

Taking the above into account, we propose to keep a separate dispatch table for each dynamically linked module (for each base method if necessary). The dynamic loader is then responsible for filling them accordingly to the static requirements of each module and the most specific overriders. This results in:

D_1	A	B
A	AA^1	AB^2
B	BA^3	AB^2

D_2	A	B
A	AA^1	AB^2
B	BA^3	BB^1

D_3	A	B
A	AA^1	AB^2
B	BA^3	BA^3

It looks as if modules now violate covariant consistency, but in reality they do not because their return types are casted back to the types that were statically expected by the modules from a call. For example in D_2 a call to foo(A&,B&) is wrapped into a thunk, that adjusts the result type from $R2$ to $R1$, so $R1$ is actually returned, which is what module expects and which is type-safe.

As can be seen, this logic may result in different functions being called for the same type tuple depending on where the call is made from. We note, however, that *the call is always made to the most specialized overrider that is type-safe for the caller*.

It is also possible that different modules provide different overriders for the same combination of types. Some of such cases can be resolved by considering covariance of the return types (§3.3). The others can be resolved by letting each module call its own implementation. In such scenario, a third module that does not provide its own implementation of that overrider will *deterministically* get one of the existing implementations, e.g. the one coming from a module with a more recent date.

6.2 Late ambiguities

Let's consider a plausible scenario involving three DLLs:

```
// dll−1
struct GuiButton { virtual ~GuiButton(); };
struct GuiEvent { virtual ~GuiEvent(); };
void handleEvent(virtual GuiButton&, virtual GuiEvent&);

// dll−2
#include<dll1>
struct MyButton : GuiButton { };
void handleEvent(virtual MyButton&, virtual GuiEvent&);

// dll−3
#include<dll1>
struct SpecialEvent : GuiEvent {}
void handleEvent(virtual GuiElement&, virtual SpecialEvent&);
```

The first DLL defines a class GuiButton, a class GuiEvent, and a base-multi-method handleEvent. Internally, a second DLL derives a new type MyButton from GuiButton and introduces a new overrider for handleEvent. Likewise, the third DLL derives a new internal class SpecialEvent from GuiEvent and introduces a new overrider. The second and third DLL could stem from different vendors that do not know about each other.

Now a call of handleEvent with a MyButton and a SpecialEvent is ambiguous. The writer of the total system (the "system integrator") should in principle have considered that possibility and dealt with it. So, one solution would be to terminate the program or to throw an exception [28]. However, such problems are hard to predict and design for. Relaxed Multi-Java [25] resolves these conflicts by introducing glue methods (to glue DLL2 and DLL3) that the system-integrator provides. While this might be a viable solution for software developers integrating several libraries, it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without direct request of a developer. This, for example, is the case with various component object models when application may ask the system to create an object with a particular name and operating system will locate and load the module it is resided in.

In §3.2.2 we saw that when dynamic type of an object cannot uniquely choose the best overrider we could use a static type of an object to disambiguate. Similarly, when module does not know anything about a particular dynamic type, because its definition was not available during compilation of the module, we may use information about its most specific static type, known at the module's compilation time. What is important, is that it must not restrict us from selecting the most specific overrider when it is available.

With this said, in the example above it seems reasonable to have calls with arguments of types MyButton& and SpecialEvent be handled by handleEvent(MyButton&, GuiEvent&) inside DLL2 and by handleEvent(GuiElement&, SpecialEvent&) inside DLL3. Having different dispatch tables per module allows us to do this. But what about calls in other modules that neither knew about MyButton nor about SpecialEvent? One option will be to treat both classes as their base classes and dispatch appropriately, but as we have just argued, static view of a module should not prevent us from choosing a better overrider. This can be supported by the fact that some modules may have only seen the interface: the base-multi-method and the roots of the hierarchies it is defined on. Nevertheless they would expect more refined overriders to handle calls on derived classes.

We note that in principle, *both* handleEvent functions should correctly handle the event; that is, both handleEvent functions must assume that its arguments could be of a further derived class that it does not know of. That is, the code of the both handleEvent functions must be written in a way that is generic on its arguments (probably using virtual functions on the individual arguments). This implies that as long as an event handler's code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, it can be replaced by the other event handler. Even a non-deterministic selection of the overrider would produce a correct result! Furthermore, we expect that two different DLLs may provide the same overrider.

With this said we propose to resolve ambiguities at load-time as following:

- If there is a unique best match among all type-safe overriders of a module that can handle a particular combination of argument types – use it.
- If there is no such a unique best match, but an overrider preferred by a static view of a module is among best matches, – it is preferred to other overriders.
- Finally, if there is no a unique best match, and the overrider preferred by a static view of a module is not among best matches, – an unspecified deterministic choice among best matches is made.

7. Related work

Programming languages can support multi-methods either through built-in facilities, pre-processors, or through library extensions. Naturally, tighter language integration enjoys a much broader design space for type checking, ambiguity handling, and optimizations compared to libraries. In this section, we will first review both library and non-library approaches for C++ and then give a brief overview of multi-methods in other languages.

7.1 Cmm

Cmm [28] is a preprocessor based prototype implementation for an open-method C++ extension. It takes a translation unit and generates C++ dispatch code from it. Cmm is available in two versions. One of them uses RTTI to recover the dynamic type of objects to identify the best overrider. The other approach achieves constant time dispatch by relying on a virtual function overridden in each class. This virtual function returns a small integer that uniquely identifies its class. Dispatch ambiguities are resolved by throwing runtime exceptions. Cmm allows dynamically linked libraries to register and unregister their open-methods at load and un-load time. In addition to open-method dispatch, Cmm also provides call-site virtual dispatch. Cmm does not provide special support for multiple inheritance and therefore its dispatch technique does not coincide with virtual function semantics.

7.2 DoubleCpp

DoubleCpp [5] is another preprocessor based approach for multi-methods dispatching on two virtual parameters. It essentially translates these multi-methods into the visitor pattern. For doing so, DoubleCpp requires access to the files containing the class definitions in order to add the appropriate accept and visit methods. DoubleCpp, like any other visitor-based approach, does not report but quietly resolve ambiguities.

7.3 Accessory Function

The accessory functions papers [15, 35] mention possible ways to enhance the basic accessory function mechanism to allow multiple virtual parameters. The compilation model they describe uses, like our approach, a compiler and linker cooperation to perform ambiguity resolution and dispatch table generation. However, the accessory functions are integrated into the regular v-tables of their receiver types, which requires the linker to not only generate the dispatch table but also to recompute and resolve the v-table index of any other virtual member function. While [15] explicitly requires an overrider to resolve ambiguities introduced by multiple inheritance, [35] adopts overload resolution rules (§3.5). The authors do not refer to a model or prototype implementation to which we could compare our approach.

7.4 Loki

The Loki [1] approach is based on Alexandrescu's template programming library with the same name. It provides several different dispatchers that balance between speed, flexibility, and code verbosity. Currently, it supports multi-methods with two arguments only, except for the constant-time dispatcher that allows more arguments. The static dispatcher provides call resolution based on overload resolution rules, but requires manual linearization of the class hierarchy in order to uncover the most derived type of an object first. All other dispatchers, including the constant time dispatcher, do not take hierarchical relations into account and effectively require explicit resolution of all possible cases.

7.5 Other languages

One of the first widely known languages to support multi-methods was CLOS [29]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity conflicts. In Cecil's [8,9] class-less object-model multi-methods are regarded as an integral part of objects. Cecil views silent ambiguity resolution as a potential source for programming errors. Therefore, it uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at runtime. In [26], the authors discuss the trade-offs between multi-methods and modular type-checking in languages with neither a total order of classes nor asymmetric dispatch semantics. Ranging from globally type-checked programs to modularly type-checked units, the models embrace or restrict the expressive power of the language to different degrees. Based on these findings, MultiJava [11] implements a model that allows separate compilation and eliminates the need for a link-time type-checker but also curtails extensibility. Relaxed MultiJava [25] re-introduces a link-time type checker and relies on the system integrator to resolve ambiguities by providing new overriders (glue-methods). An example of a language adding multi-methods through a library is Python [32]. Chambers and Chen [10] present an alternative implementation technique based on a lookup DAG. Their work generalizes multiple dispatch to be a subset of predicate based dispatch.

8. Results

In order to discuss time and space performance, we implemented the Shape-intersection example for regular C++ (Visitor), our open-methods, multi-methods, the two Cmm branches (default and experimental constant time), and the LOKI library.

We wrote 20 classes (representing shapes, etc.) which can intersect each other. All in all, this results in 400 combinations for binary dispatch functions. We implemented 40 specific intersect functions to which all of the 400 combinations are dispatched to. In order to get a reliable timing of the function invocation, these 40 intersect functions only increment a counter. Since not all techniques we use support multiple inheritance, these 20 classes only use single inheritance. The actual test consists of a loop that randomly chooses two out of 32 objects and invokes the intersect method. We implemented a table-based random number generator that is simple

and does not contain any unpredictable operation such as floating-point calculations or integer-divisions. We ran the loop twice with the same random numbers: The first run allows implementations, which build the dispatch data structure on the fly to warm up and load data/code into the cache. The second loop was timed. The clock-cycle based timer takes the time before and after the loop and we calculate the average number of clock-cycles per loop to compare the results.

8.1 Implementations

We used g++ 4.0 and compiled our test-code with optimization -O3. We ran our test code on a dual-core (Pentium D, 2.8Ghz) under CentOS Linux from the login-shell. We used nice -20 to invoke our test-programs with the highest possible priority.

C++ Visitor The implementer of the visitor has to foresee all possible shapes and provide interfaces for it. For example:

```
struct Shape {
  virtual void intersect(Shape& shape);
  virtual void intersect(Rectangle& shape);
  virtual void intersect(Circle& shape);

  virtual void accept(Shape& shape) { shape.intersect(*this); }
};
```

The concrete first type is recovered by `accept()`; the second type is recovered by `intersect()`.

C++ Open-Methods/Multi-Methods This approach is based on the object model described in §4.

Chinese Remainder Using the Chinese Remainder approach (§4.3.2), the number associated with the dispatch table grows exponentially with the number of types, we could only implement a simplified version that can handle eight types instead of 20. Hence, we omit the size of the program executable.

Loki Only the static dispatcher was used in our tests with Loki since other dispatchers require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of manually allocated array of functions through which calls are made. However, as we indicated before, dual nature of multi-methods require them to provide both dynamic dispatch and automatic resolution mechanism.

Other approaches The Cmm versions and DoubleCpp are described in §7.1 and 7.2, respectively.

8.2 Results & Interpretation

Our experimental results can be summarized in terms of execution time and program size:

Approach	Program size	Cycles/Loop
Virtual function	n/a	75
C++ Multi-method	19 547	78
C++ Open-method	19 725	82
Double Cpp	20 859	120
C++ Visitor	35 289	132
Chinese Remainders	n/a	175
Cmm (constant time)	112 250	415
Cmm	111 305	1 320
Loki Library	34 908	3 670

8.2.1 Executable size

The size of dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as programming language feature [35]. However, our results reveal that the C++ multi-method based approach is 80% smaller than the visitor approach that is based on a brute force implementation. Each class implements intersect methods for all 20 types of the hierarchy. A somewhat smarter approach would be to remove redundant intersect overriders. However, removing specific overriders is tedious and difficult to maintain, since the dispatch would be based on the static type information of the base class. Intrinsically, DoubleCpp reduces the program size for visitor based implementations. Nevertheless, the optimized result would be at best able to match the multi-method implementation, simply because each type contains 20 intersect entries in the v-table. Multiplying this with the number of shapes, 20, results in 400, exactly the number of entries found in the dispatch table. We do not discuss the program size of the two Cmms and Loki, since they use additional header files such as <typeinfo> and <stdexcept> that distort a direct comparison.

8.2.2 Execution time

Both Multi-Methods and Open-Methods are (as expected) roughly comparable to a single virtual function dispatch, which needs 75 cycles per loop. Hence, the better performance compared to the visitors is not surprising. However, the fact that multi-methods reduce the runtime to 62% of the reference implementation using the visitor is noteworthy. We conjecture this is an effect of the size of the class hierarchy and that double dispatch depends on the number of overriders. Our conjecture is supported by two observations: firstly, the DoubleCpp-based visitor eliminates redundant overriders and runs slightly faster. Secondly, we also simulated an analysis pass dispatching over AST-objects of 20 different types and counting the category to which they belong (type, declaration, expression, statement, other). In this case, the double dispatch has only 20 leaf-functions instead of 400 and our dispatch test runs 78 cycles instead of 132. The open-method approach (requiring only five overriders), is still faster and needs 68 cycles.
The difference between multi-methods and open-methods is within the expected range. Three more indirections require 4 more clock cycles. Although significantly slower, Cmm (constant time) performs better than expected, since its author estimates the dispatch cost as 10 times a regular virtual function call. As expected the two non-constant time approaches perform worst.

8.2.3 Significance of performance

The performance numbers comes from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application the improved performance might or might not be significant. For the image conversion example, gains in execution speed are negligible compared to time spent in the actual conversion algorithm. In other cases, such as the evaluation of expressions using user-defined arithmetic types, traversal of abstract syntax trees, and some of the most frequent shape intersect examples, the speed differences among the double dispatch approaches appear to be notable.

Contrary to much "popular wisdom", our experiments revealed that for many applications the use of dispatch tables for open-methods and multi-methods actually reduce the program size compared to brute-force and work-around techniques.

9. Conclusions and future work

We presented a novel approach to dispatching open multi-methods that is in line with the multiple inheritance semantics of the current C++ object model and the C++ overload resolution rules. This implies compile-time or link-time detection of ambiguities. By considering covariant return type in the ambiguity resolution we reduce the number of potential conflicts. We have discussed an implementation based on modifications to the EDG compiler front-end and have described a mechanism that supports the integration of several

translation units. Our evaluation of different approaches to implementing open-methods in C++ shows that our approach is significantly better (in time and space) than current alternatives. Indeed, it is almost as efficient as single dispatch. Because the dispatch is constant time and does not rely on exceptions to signal ambiguities, it is applicable in embedded and hard real-time systems.

Planned future work includes:

9.1 Virtual Function Templates

Virtual function templates are a powerful abstraction mechanism. However, C++ cannot do that because generating v-tables for virtual function templates requires a whole-program view and C++ traditionally relies almost exclusively on separate compilation of translation units. The pre-linker technique described here should be able to synthesize v-tables for virtual function templates as it does for open-methods.

9.2 Function pointers to open-methods

Pointers to member functions in C++ preserve polymorphic behavior when they point to a virtual member function. To be in line with this semantics, pointers to open-methods should preserve dynamic dispatch too. This could be implemented by allocating a proxy function every time an address of an open-method is taken and returning address of this proxy. Inside the function compiler simply generates call to appropriate open-method. Note that similar to single dispatch in C++, it is not possible to take an address of a particular open-method overrider, - returned pointer will always have a polymorphic behavior.

9.3 Calling a base implementation

C++ provides a syntax to call a particular overrider of a virtual member function directly, avoiding dynamic dispatch. This is often used to call the function in the base class. To do this, C++ requires the user to use a fully qualified name of virtual member function: e.g.: `p->MyClass::foo()`; It is likely that similar functionality will be required for open-methods. We would have to invent some syntax for fixing the type of either individual or all arguments.

9.4 Space Optimizations

With class hierarchies consisting of around 100 classes, the size of dispatch tables can become significant, especially when we consider support for covariant return types. Several techniques to compressing and reusing of dispatch tables have been proposed in [2]. Proposed techniques should be directly applicable to our approach so we would like to implement them in the future.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.

[2] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA '94: Proceedings of the ninth annual Conf. on Object-oriented programming systems, language, and applications*, pages 244–258, New York, NY, USA, 1994. ACM Press.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 3rd edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] L. Bettini, S. Capecchi, and B. Venneri. Double dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.

[6] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula BEGIN*. Auerbach Press, Philadelphia, 1973.

[7] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, 1995.

[8] C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92: Proceedings of the European Conf. on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[9] C. Chambers. The Cecil language: Specification and rationale. 3.2. Technical report, Department of Computer Science and Engineering. University of Washington, 2004.

[10] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 238–255, New York, NY, USA, 1999. ACM Press.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[12] Codesourcery.com. The Itanium C++ ABI. Technical report, 2001.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[14] Edison Design Group. C++ Front End., March 2006.

[15] C. B. Flynn and D. Wonnacott. Reconciling encapsulation and dynamic dispatch via accessory functions. Technical report, 1999.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36(2):139–156, 2006.

[18] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[19] D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming. *OOPSLA'06*, 2006.

[20] International Organization for Standardization. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. pub-ISO, pub-ISO:adr, 1994.

[21] ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.

[22] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press.

[23] Lockheed Martin. *Joint Strike Fighter, Air Vehicle, C++ Coding Standard*. Lockheed Martin, December 2005.

[24] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[25] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN Conf. on Object-oriented programing, systems, languages, and applications*, pages 224–240, New York, NY, USA, 2003. ACM Press.

[26] T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conf. on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag.

[27] A. Shalit. *The Dylan Reference Manual. 2nd edition*. Apple Press, 1996.

[28] J. Smith. Draft proposal for adding multimethods to C++. 2003.

[29] G. L. Steele Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[30] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[31] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[32] G. van Rossum and F. L. Drake, Jr. *Python Language Reference Manual*. 2005.

[33] J. Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282, New York, NY, USA, 2001. ACM Press.

[34] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety prooffor multiple inheritance in c++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 345–362, New York, NY, USA, 2006. ACM Press.

[35] D. Wonnacott. Using accessory functions to generalize dynamic dispatch in single-dispatch object-oriented languages. In *COOTS*, pages 93–102. USENIX COOTS, 2001.

[36] www.fourcc.org. Video codec and pixel format definition, February 2006.