An Empirical Study of Function Overloading in C++

Cheng Wang and Daqing Hou
Clarkson University
Potsdam, NY USA 13699
dhou@clarkson.edu

Abstract

The usefulness and usability of programming tools (for example, languages, libraries, and frameworks) may greatly impact programmer productivity and software quality. Ideally, these tools should be designed to be both useful and usable. But in reality, there always exist some tools or features whose essential characteristics can be fully understood only after they have been extensively used. The study described in this paper is focused on discovering how C++'s function overloading is used in production code using an instrumented g++ compiler. Our principal finding for the system studied is that the most 'advanced' subset of function overloading tends to be defined in only a few utility modules, which are probably developed and maintained by a small number of programmers, the majority of application modules use only the 'easy' subset of function overloading when overloading names, and most overloaded names are used locally within rather than across module interfaces. We recommend these as guidelines to software designers.

1 Introduction

Programming languages are one of the many important tools that programmers use in their daily problem solving. With the ever increasing size and complexity of the problems tackled by computer software rises the sophistication of programming tools. This is a bad news for the average programmers because to do their jobs properly, they must master their tools first, which will require more effort when the tools grow bigger and become more complex. In particular, current programming languages are providing programmers with an increasingly large set of features to support them in solving diverse problems. Consequently, mastering a whole programming language demands serious investment on the part of the programmers. [1] While it is relatively easy to learn how to write and compile small pieces of example code that demonstrate the use of a particular language feature, it is challenging for the programmers to decide what features to learn that can solve their problems at hand in the best possible way. It would be ideal if for each language feature, there is a methodology defined that guides the programmers on what kind of a problem the feature is useful for and how to solve the problem properly with the feature [11].

The formulation of such a methodology could be best done by examining and reflecting on as much as possible empirical evidence from the serious use of the particular language feature of interest. While language designers tend to do their best in guiding their design process with practical experience and user feedback [2, 11, 14], not all design decisions are, or can be, made based on solid empirical evidence. Sometimes, there simply is not a sufficient amount of experience available at the time a particular design decision is made [14]. For sophisticated features like overloading and templates, while there is no doubt that such a feature as a whole is useful, its interactions with other features can be numerous and the details can be subtle to pin down. Sometimes, decisions can be made in an arbitrary and ad hoc fashion, or based on particular requirements that are controversial (for example, the decision for C++ to support implicit narrowing in overloading resolution in order to maintain compatibility with C). For all of these reasons, there is always a need to examine how language features are *actually* used in the wild in order to understand the consequences of such decisions, to identify typical use cases of language features, and to develop guidelines.

One way to gather such empirical evidence is by analyzing the source code of large-scale systems. We chose to study the use of the compilation-time overloading feature in C++ because it involves a large set of rules that many programmers would not be confident with. Furthermore, C++ overloading interacts with many other features of the language, for example, namespaces, inheritance, and tem-

[1] Recognizing this problem of increasing language scale, some language designers are trying to develop a minimal core language and shift some of the complexity to external libraries and APIs. However, we believe that learning such a minimal language can still be a challenge for the average programmers.

```
class Y;
class X {
public:
operator char() const {return 'a';}
void foo (int); //f1
void foo (char);//f2
void foo (double);//f3
void foo (X);//f4
void foo (Y&);//f5
};

class Y: public X { };

void foo (double); //f6
void foo (int);//f7

void bar(Y & aY){
//C={f6,f7}, V={f6,f7} best:
//f7<cr_promotion:ck_std>
foo('c');
//C={f6,f7},V={f6,f7} best:
//f7 <cr_user:ck_user->ck_std>
foo(aY);
//C={f1...f5},V={f1,f2,f3} best:
//f2 <cr_std:ck_ptr,cr_identity:ck_identity>
aY.foo('a');
//C={f1...f5},V={f1...f5} best:
//f5 <cr_std:ck_ptr,cr_identity:ck_ref_bind>
aY.foo(aY);}
```

Figure 1. Example of function overloading and the process of overloading resolution. C stands for candidate set and V for viable set.

plates. Thus we felt that overloading would be a complex feature for which a methodology is needed to guide its use. There is also doubt on the usefulness of compilation-time overloading [11]. We hoped that our study could shed some light on this controversy as well.

In order to gather data about the use of overloading in practical systems, the GNU g++ compiler [2] is modified and used to compile a Mozilla browser. [3] In this paper, we analyze the data and report our findings about the use of overloading in the Mozilla code base. Analysis of data extracted from another system (MySQL [4]) is currently under way.

The rest of this paper is organized as follows. Section 2 provides a brief overview of C++ overloading. Section 3 describes the study method, how g++ is instrumented, and what kind of data are gathered. Section 4 presents the results of our analysis. Section 5 presents related work. Section 6 concludes the paper and anticipates future work.

2 Overloading in C++

Overloading allows a single name to be used to represent more than one operation. It is a compile-time feature. That is, given an expression that applies an overloaded name, a compiler needs to choose a single candidate that best matches the expression (overloading resolution). The compiler thus needs to implement a set of rules that govern

1. how to identify from the surrounding scopes of the expression the set of candidate operations with the same name as what is used by the expression (candidate set),

2. how to convert from argument types appearing in the calling expression to the parameter types in the definition of a candidate operation and rank the conversions,

3. how to determine the set of operations to whose parameter types there exist feasible conversions from types in the expression (viable set), and

4. how to determine the best candidate from the viable set as the final result that the expression should be linked to. If no such candidate can be found, the process fails and a compilation error is given to the programmer.

Table 1 summarizes the implicit type conversion rules of C++. A rank on top of the table is considered better than ones below it. Also note that a rank may contain subcategories known as kind.

Figure 1 shows a simple example designed to illustrate the process of overloading resolution in C++. For each overloaded call in `bar()`, its candidate set and viable set, the best candidate, and the conversion sequences from arguments to parameter s are given in the comment above the call. For example, `foo('c')` has a candidate set that contains `f6` and `f7`. Its viable set also contains `f6` and `f7` since there are conversions from `char` to both `double` (for `f6`) and `int` (for `f7`). The conversion from `char` to `int` has a rank of `cr_promotion` and a kind of `ck_std`. which is represented as `<cr_promotion:ck_std>` in the comment. Note that in this example, `foo(aY)` has a conversion sequence of length 2, and the conversion sequences of all other calls are of length 1.

Overloading, when used appropriately, can increase program readability and clarity [10]. Without this feature, it would be rather difficult to designate different but meaningful names to the several `foo`'s in Figure 1 when the operations have similar behavior but different parameter types.

On the other hand, the size of Table 1 indicates that C++ overloading rules are rather complex to understand. But this is the result of a careful design that accommodates multiple design requirements of C++ [14]. Each conversion rule has a reason. For example, `ck_std` results from the design decision to maintain compatibility with C's basic types and conversions (narrowing), and `ck_ptr` from the interaction with inheritance. `cr_user` is designed to make it possible for user-defined types to be used in the same way as primitive types. In particular, with the addition of operator

[2] http://gcc.gnu.org/
[3] http://www.mozilla.org/
[4] www.mysql.com

rank	kind
cr_identity	ck_identity (two identical types), ck_lvalue (e.g., array to pointer), ck_rvalue (l-value to r-value), ck_ref_bind (from a type to reference to the type).
cr_exact	ck_qual (adds a qualifier like const to the base type in a pointer type).
cr_promotion	safe conversion from shorter data type to longer one (widening).
cr_std	ck_std (narrowing), ck_ptr (conversion to pointer to the most 'derived' base type), ck_base (conversion to the most 'derived' base type), ck_pmem (conversion to pointer to member function). But a conversion of ck_std may be given a rank of cr_promotion if, for example, in a conversion from enum to int of kind ck_std, the integral value of the enum is known to fall within the range of int.
cr_pbool	An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type bool.
cr_user	ck_user (user-defined conversion, either via a constructor or a type conversion operator).
cr_ellipsis	Conversion to ellipsis in target type.
cr_bad	No conversion possible.

Table 1. C++'s implicit type conversion rules as implemented by g++.

overloading, it becomes possible for a user-defined type to work immediately with generic algorithms and data structures without change.

Some advanced overloading features, like operator overloading and user-defined conversions, may hinder program understanding. For example, when resolving `foo(aY)`, in addition to knowing the type of `aY` and the existence of 2 `foo`'s found in the global scope, a programmer also needs to visit the body of class `Y` and class `X`. In extreme cases, the resolution of an overloaded operator may even require the visit of as many as 7 scopes [10].

Given the above analysis, it appears reasonable to propose the following guidelines for the use of overloading.

- If a name is going to be used only a few times by a client, use overloading only if absolutely necessary and make it as easy as possible for the client to resolve an overloading call among the candidates (for example, by designing the candidates such that they have obviously different parameters).
- When a name is going to be used many times by a client, since the benefit of using overloading may outweigh its cost, its use is justified.
- User-defined conversion operators should be used only when they are going to be used many times. Control the scope of their use. Consider using `explicit` to prevent single-parameter constructors from being used inadvertently as type convertors.

But there are still questions to be answered: What is overloading used to achieve? How are the various overloading rules used in practice? How are user-defined conversions used? In the next sections, we try to answer these questions by studying the use of overloading in Mozilla.

3 Study Method

To gather data for the use of overloading in large-scale systems, we decided to instrument the GNU C++ compiler g++ and use it to compile open-sourced software. Our first test case is Mozilla. In this section, we describe how g++ is instrumented, the data schema for the data we extracted for the definition and use of function overloading. We also characterize the Mozilla code base.

3.1 Instrumenting g++

Internally, g++ uses a tree data structure to represent program elements and symbol tables. It also provides a rich set of macros for traversing the tree structure and accessing attributes of individual tree nodes.

The *definitions* of overloaded functions and operators are obtained by hooking into the name resolution process of g++. Specifically, when a new function is encountered, the compiler will be able to conclude whether it is overloaded with any other functions that it has seen previously. The fact that a function is overloaded is intercepted and the information about the group of overloaded functions is stored in a data structure made by us.

In our data structure, a group of overloaded functions is identified by a fully qualified name, absolute paths for the files where the overloaded functions appear (in rare cases, overloaded functions may originate from different header files), and the path for the file where the group becomes effectively overloaded. After the compilation of a translation unit is completed, the information about all overloaded names that occur inside the translation unit is persisted, one line per name, into a text file, which is given a name that is distinct to the translation unit. For example, the following line of text captures the fact that the global operator `new`

defined in the standard C++ header file `new` is included in a file named `nsXFontNormal.cpp`, and the operator is overloaded 3 times. Fields are separated by the # sign.

```
::operator new#/include/c++/4.2.0/new#nsXFontNormal.cpp#3
```

The *calls* of overloaded functions and operators are obtained by hooking into the type checking process where the type of an expression is resolved. [5] Specifically, given an expression like a function call, g++ will indicate which function or operator the expression should be linked to, and whether the function or operator is overloaded. If an overloaded function is involved, the expression is resolved by selecting from multiple candidates the one that matches the best by following the set of overloading rules of C++. A list of implicit conversions is produced, which indicates how the types of the arguments can be converted to the parameter types in the function being linked to.

For each overloading call, the following record is produced as a line in a text file:

- a fully qualified function name that the current expression is resolved to,
- absolute file path and line number where the winning function is defined,
- absolute file path and line number where the function is called,
- candidate set size,
- viable set size, and
- list of ranks, kinds, and from and to types for argument to parameter conversion.

The text files generated by running the instrumented g++ compiler are then processed with awk scripts and other Unix utilities like `sort` and `uniq` to obtain the desired statistics.

3.2 Study subject: Mozilla

We chose Mozilla as the first subject of our study because it is an open-sourced, large-scale system written in C++. We used version 1.8b1. Table 2 shows some static measures of its size before compilation to give an impression of its scale.

#header files	4679	#html files	2246
#cpp files	4442	#xul files	624
#c files	1515	#xml files	325

Table 2. Some measures of Mozilla 1.8b1 size.

Table 3 lists a subset of Mozilla modules for which the following module dependency exists. Modules xpcom, intl, and nspr make up the utility layer. Modules netwerk, widget, view, gfx, and rdf provide various systems services, with modules widget and view depending on gfx. Modules content and parser handle documents like HTML files. Module layout sits on top of all of the above modules. Module xpfe provides platform-portable implementation for browser. Finally, the top-most layer contains modules browser, editor, dom, and js.

module name	description
browser	Mozilla web browser
editor	a.k.a composer
dom	Document Object Model
js	JavaScript
xpfe	cross-platform front end
layout	APIs for laying out UI
content	modeling contents like HTML and XML
parser	HTML parser
netwerk	networking APIs
gfx	graphics APIs
widget	
view	
rdf	Resource Description Framework
nspr	netscape portable runtime
xpcom	cross-platform component model
intl	internationalization

Table 3. A subset of Mozilla modules.

The xpcom module needs to be mentioned because it makes extensive use of overloading. xpcom provides APIs for implementing the COM component model. It also provides a rich set of classes for string manipulation, a template class `nsCOMPtr` for implementing a smart pointer, as well as support for threading, hash tables, and arrays.

Given a pointer `p` to a class `X`, a smart pointer of type `nsCOMPtr<X*>` can be created and used on behalf of `p`. In places where an `X*` is needed, `nsCOMPtr<X*>` is converted to `X*` automatically. The following code snippet illustrates how the conversion happens. Specifically, the user-defined conversion operator defined in `nsCOMPtr` will first convert `nsCOMPtr<X*>` to a pointer to `nsDerivedSafe<X*>`. Because `nsDerivedSafe<X*>` is a subclass of class `X`, the pointer is then converted to a pointer to the most derived base type, which is `X`. Note that in this case, both the definition and the use of user-defined conversion is confined within the same module xpcom and can be used by a client without knowing the exact details of the conversions.

```
template <class T>
class nsDerivedSafe: public T { ... }

template <class T>
class nsCOMPtr {
    operator nsDerivedSafe<T >* () const {...}
}
```

[5] Inlining would have no impact on the completeness of our data as it happens after type checking.

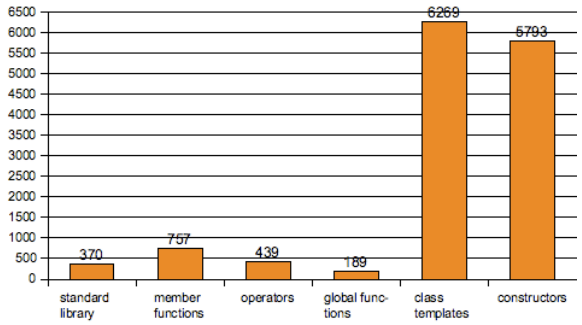

Figure 2. Distribution of 13817 overloaded function names over categories.

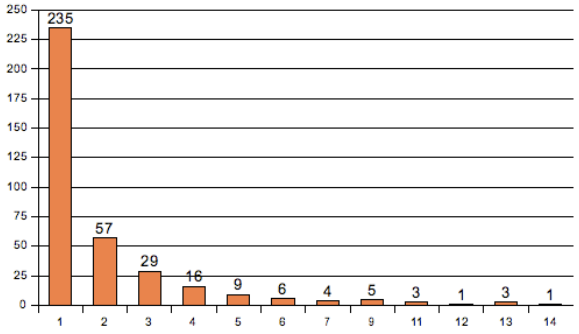

Figure 3. Distribution of 375 classes over the number of function names overloaded by a class.

In Mozilla, files belonging to the same module are stored into the same file system directory. Therefore, based on the file path our tool extracts for each translation unit, we are able to obtain information about the module where an overloading call is made as well as the module that defines the function to which the overloading call is resolved.

Mozilla 1.8b1 is configured to build a Mozilla browser and is compiled with the instrumented g++ compiler. In this way, both the definition and the use of overloaded functions are obtained and the data are stored in text files, which are further processed to produce the desired statistics.

4 Results

This section presents the characteristics of the definition and use of overloading in the Mozilla code base.

4.1 Overall statistics of definition of function overloading in Mozilla source

A total of 13 817 overloaded names are detected. Figure 2 depicts the distribution of overloaded names over the

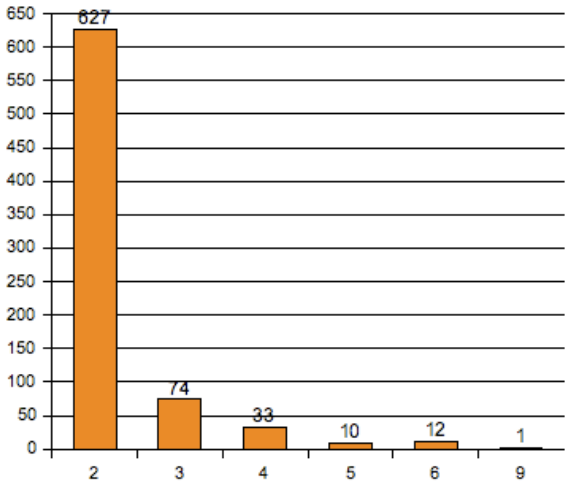

Figure 4. Distribution of 757 class members over the times they are overloaded.

categories of standard library (370), instantiations of class templates (6269), operators (439), global functions (189), constructors (5793), and member functions (757). The vast majority of overloading is due to constructors and class template instantiations (42% and 45%, respectively). In particular, the 6269 overloaded names in the category of class template instantiations are actually caused by only 11 names from 6 template classes, all from the module xpcom, which are shown in Table 6.

In compiling the Mozilla browser, 5 689 classes are detected (excluding those contributed by instantiating class templates and the standard C++ library). Out of these, 375 classes overload at least one function name. That is, about 6.6 percent of the classes define overloaded member functions. Figure 3 depicts the distribution of 375 classes over the number of names overloaded by each class. The classes that contain 14 and 13 overloaded names are `nsIRenderingContext` and its 3 subclasses in the gfx module. The `morkStore` class from module db overloads 12 names. The xpcom module contributes 3 classes for 11 names and 5 classes for 9 names, 6 of which are string classes, and 2 for COM implementation.

Figure 4 depicts the distribution of the 757 overloaded member functions over the number of times a name is overloaded. To understand for what purposes certain names are overloaded a high number of times, we inspected the 23 names that are overloaded 5 times or more. We found that 16 are string operations, and 5 are graphics operations (`GetWidth` from 4 classes and `DrawString`). The name that is overloaded 9 times is `nsINodeInfo::Equals`, which are used to compare between objects of various types.

Table 4 depicts those Mozilla modules that contain more than 10 overloaded names in member functions and the ac-

editor/composer	27	layout	104
htmlparser	31	dom	11
content	129	netwerk	14
gfx	144	intl	14
view	11	xpcom	113
widget	17	js	13

Table 4. Mozilla modules that contain more than 10 overloaded names in class scope.

tual number of overloaded names that each such module contains. To gain some insights into where and how names are overloaded, we inspected a subset of the 757 overloaded names. In general, it appears that overloading tends to be used in modules that manipulates data structures, for example, content for document structures, gfx for geometric shapes and drawing, layout for data structure traversal and layout logic, and xpcom for strings and pointers. Our informal inspection also reveals three patterns for function overloading. One pattern is to overload getters and setters to provide different ways of setting and getting object attributes. Another is to overload a core operation with several others that are reduced to the core one. Yet another pattern is to provide two ways of retrieving object attributes, via return values and via a parameter, respectively.

In sum, these data indicate that only 6.6 percent of classes (375) in Mozilla overload member names, 85.6 percent of these classes overload 3 or less members, and 92.6 percent of the overloaded members (757) are overloaded 2 or 3 times (82.8 percent are overloaded only 2 times).

4.2 Overall statistics of calls of overloaded functions in Mozilla source

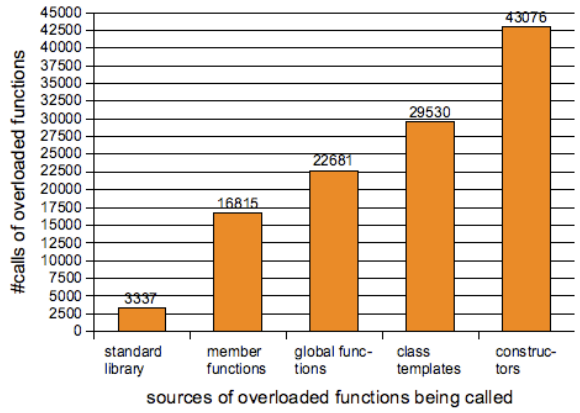

Figure 5. Distribution of 115439 calls of overloaded functions over categories.

A total of 115 439 function calls are identified as calling overloaded functions. Figure 5 depicts the distribution of these function calls. Of these, 29 530 are due to class templates (no template functions are called). 3 337 are calling overloaded function defined in the standard C++ library. The remaining 82 572 calls are of non-library, non-template overloaded functions.

::operator delete	::operator delete []
::operator new	::operator new []
std::abs	std::div
std::memchr	std::strchr
std::strrchr	std::strstr
std::strrpbrk	

Table 5. The 11 overloaded functions in the standard C++ library used by Mozilla.

Table 5 depicts the 11 overloaded function names in the standard C++ library used by Mozilla. As an example of the intricacy of C++ overloading rules and what it means for C++ to maintain compatibility with C, consider the following code snippet that depicts how C++ overloads the C function `strchr`. Its purpose is to correct a type problem in the C prototype for `strchr`. Since the C `strchr` returns a pointer to the content of its first argument, which is of type `const char *`, according to C++'s typing rules, its return type should have been `const char*` too. To correct this problem and to maintain maximum compatibility with C, another `strchr` is added in C++, which overloads the C `strchr`. Note that the 2 functions are distinguished by the `const` keyword. This example demonstrates only one of the many subtle rules C++ defines for function overloading.

```
//in string.h (C header):

char *
strchr(const char*, int)

//in cstring (C++ header):

using ::strchr;

inline char*
strchr(char* s1, int n)
{return _builtin_strchr(const_cast<const char*>(s1),n);}
```

In the 29 530 calls that are resolved to an overloaded member in a class template (see Figure 5), 23 387 are calls of template constructors, and 6 143 calls of overloaded member functions in template instantiations. Further analysis reveals that these member functions overload 11 names in 6 template classes, all from the xpcom module. These 11 names are depicted in Table 6. Note that the `nsCOMPtr` class template, which implements a smart pointer, is widely used (4881 times in 27 modules). Note also that 8 of the 11 overloaded names are operators.

overloaded name	#calls (#module)
nsReadingIterator<T>::operator++	198 (13)
nsReadingIterator<T>::operator- -	30 (10)
nsWritingIterator<T>::operator++	9 (2)
nsWritingIterator<T>::operator- -	3 (1)
nsAutoArrayPtr<T>::operator=	4 (2)
nsAutoPtr<T>::operator=	89 (4)
nsCOMPtr<T>::get_address	711 (1)
nsCOMPtr<T>::operator=	4881(27)
nsCOMPtr<T>::swap	22 (8)
nsRefPtr<T>::operator=	188 (8)
nsRefPtr<T>::swap	8 (3)

Table 6. The 11 overloaded template member functions (all defined in xpcom) and the distribution of 6143 calls of these functions over modules.

In the 82 572 calls of non-library, non-template overloaded functions, 43 076 of them are calling class constructors, 22 681 are calling global functions, and 16 815 are calling class member functions (see Figure 5). At this stage of compilation, internally g++ names all constructors uniformly as `base_ctor` and `comp_ctor`. This makes it easy to remove constructor calls from the data and keep only the calls of global functions and class members. As part of its building process, Mozilla performs a variety of tests on the g++ compiler to see if the compiler is suitable for building Mozilla. A few hundreds of overloading calls are generated due to these tests, which do not belong to Mozilla. A further cleanup is done to remove these calls, and a total of 39 012 overloading calls are obtained, which are non-library, non-template, and non-contructor calls in Mozilla.

4.3 Detailed analysis of overloading calls

candidate	#calls	#name	viable	#calls	#name
2	12120	649	1	30448	692
3	6106	92	2	7084	174
4	13600	44	3	1187	44
5	2976	22	4	167	21
6	3052	20	5	30	8
8	936	4	8	52	2
9	222	3	9	44	1

Table 7. Distribution of 39012 overloading calls over candidate set sizes (column candidate) and viable set sizes (column viable).

functionality	overloaded names
string and character	::Substring, ::ToLowerCase, ::ToUpperCase, nsAString:: (Append, Assign, operator+=, operator=) ... (26 more elided)
file and stream	nsFilePath::operator=, nsFileSpec::operator=, nsFileURL::operator=, nsOutputFileStream::operator<<, nsOutputStream::operator<<
data and graphics	nsINodeInfo::Equals, nsIRenderingContext::GetWidth, nsRenderingContextGTK::GetWidth, nsRenderingContextPS::(DrawString, GetWidth), nsRenderingContextXlib::GetWidth, ::FindInReadable, ::address_of, nsMetaCharsetObserver::Notify

Table 8. A total of 49 overloaded names have a candidate set size from 5 to 9.

functionality	overloaded names
string and character	nsAString::(Assign, operator=), nsAutoString::operator=, ... (14 more elided)
file and stream	nsFilePath::operator=, nsFileURL::operator=, nsFileSpec::operator=, nsOutputFileStream::operator<<, nsOutputStream::operator<<
data	Value:: (operator!=, operator==, Equals, operator=), nsGlobalHistory::SetRowValue

Table 9. A total of 32 overloaded names have a viable set size from 4 to 9.

In this section, overloading calls are analyzed in detail along three dimensions: the sizes of candidate sets and viable sets, their distributions within and across module boundaries, and how inter-module overloading calls use the conversion rules. The goal of this analysis is to understand how overloading resolution may influence programming understanding and discuss its implications on the use of various rules about overloading.

Table 7 depicts the distribution of the 39012 calls over the various sizes of candidate set and viable set. For example, the first row shows that 12120 calls have a candidate set of size 2 and are the result of calling 649 overloaded names. Furthermore, 30448 calls have a viable set of size 1 and are the result of calling 692 overloaded names. Note that 77.8 percent of all the overloading calls (30448) have a viable set of size 1, and that 81.6 percent of calls have

	editor	layout	content	parser	netwerk	gfx	widget	view	xpcom
editor	*248*	13	28	1	3		4		**3339**
layout		*1534*	149		14	478	13	9	**2553**
content		76	*1409*	9	60	7	6	1	**5947**
parser				*393*	4				**381**
netwerk					*133*				**2299**
gfx		2				*334*			**496**
widget		1		1	3	12	*36*		**362**
view						27	6	*58*	**43**
xpcom									*3387*

Table 10. Distribution of 39012 overloading calls over module interactions. Calls within modules are made *italic* and calls to xpcom bold.

a candidate set of size 4 or less. In general, this is a good news because it means that the majority of overloading calls are made with almost no extra cognitive effort from the programmers. Programmers would be able to easily recognize the function intended to be called from other candidates because the function being called has either a different number of arguments than that of the parameters of the candidates or obviously incompatible types from that of the candidates for which no implicit conversions are possible.

To gain some insights into the question as to why some names are overloaded more times than the 'good' ones discussed above, names resulting in large candidate sets and viable sets are collected and analyzed. Table 8 depicts the 49 names that are involved in calls that have a candidate set size from 5 to 9. Table 9 depicts the 32 names that are involved in calls that have a viable set size from 4 to 9. Note that the two tables share more than 20 names, most of which are string and file operations. 22 of the overloaded names in Table 9 are operations on string and character, and file and stream from the utility module xpcom, which programmers are likely to be familiar with due to their generic nature. In fact, only 10 overloaded names with a viable set size from 4 to 9 are contributed by application modules. In particular, the one with a viable set size of 9 is an operator (`nsOutputStream::operator<<`) and is called 44 times.

Table 10 depicts a matrix for the number of overloading calls that modules in the left-most column made to modules in the top row. The diagonal cells represent the overloading calls within each module, and a non-diagonal cell represents the interaction between modules. First note that 84 percent of overloading calls (32717 out of 39012) are made on 182 names in the utility module xpcom. Thus each overloaded name in xpcom is used 180 times in average. Also noteworthy is that for modules other than xpcom, within-module calls far out-number interactions between modules. This is likely to be a good attribute because modules tend to be owned by individuals or small teams that work closely with each other, which in general should facilitate the use of overloading.

There are only 1332 (3.4 percent) inter-module overloading calls among application modules (as opposed to the utility module xpcom). These calls are made to 86 overloaded names defined in 7 modules, of which 8 are global functions, and 78 class members. No operators are involved. The gfx module contributes 44 names, content comes next with 14 names, and widget and view 1 each. In average, each module contributes 12 names, and a module uses a name 6.9 times. Furthermore, each name is used in average by only 2 external modules (with a range 1 to 20). This means that application modules tend to define a small number of overloaded names, which are in turn used by only a small number of other application modules. This is clearly different from utility modules, which may define a large number of overloaded names that are used by a large number of application modules.

conversion	**number**
`cr_identity`	2568
`cr_exact`	100
`cr_promotion`	5
`ck_std`	48 (int vs unsigned int), 158 (0 to pointer)
`ck_ptr`	238 (225 due to a template in xpcom)
`ck_base`	316 (all due to strings in xpcom)
`cr_user`	379 (all due to types defined in xpcom)

Table 11. Distribution of 3812 conversions for the 1332 inter-module overloading calls.

Since overloaded names on module interfaces tend to be defined and used by different developers who are most likely working on different aspects of the application, it would be a good strategy to minimize the use of overloading on module interfaces, and to use the easy subset of overloading only. The 1332 inter-module overloading calls are analyzed to understand in detail how the conversion rules of Table 1 are used. A total of 3812 pairs of conversion between argument type and parameter type are identified. Thus in average each overloading call requires less than 3 conversions. Table 11 depicts the result. Notice that the uses of the easy rules, `cr_identity`, `cr_exact`, and `cr_promotion`, occupy more than 70 percent of the 3812 conversions. The data for the standard conversion are broken down into 3 sub-categories. 48 of ck_std convert between integral values, which may lose precision, and 158 convert 0 to a pointer type. 238 conversions are from pointer to derived type to a base type, of which, 225 are due to a template defined in xpcom (`nsDerivedSafe<T>`). 316 derived-to-base-object conversions are between the various string types defined in xpcom. Finally, the 379 user-defined conversions are implemented by 3 templates (`nsCOMPtr<T>`, `nsGetterAddRef<T>`, and `nsRefPtr<T>`) and 3 classes in xpcom (`nsCStringTuple`, `nsXPDILCString`, and `nsGetterCopies`). The 3 templates implement the so-called 'smart pointer', and the 3 classes are about strings and 'smart pointer' as well.

In sum, the majority of overloading calls in Mozilla have a viable set of size 1 (77.8 percent) and a candidate set of size 4 or less (81.6 percent). This is good because it would imply that the programmers are benefiting from overloading at almost no extra cost in performing overloading resolution. Furthermore, calls with a viable set size larger than 3 and a candidate set larger than 4 involve only 10 names from application modules, which would imply that the cost associated with these uses of overloading can be managed. Only 3.4 percent of overloading calls (1332) happen between application modules, and the rest are calls to the utility module xpcom and within-module overloading calls, which are considered less a problem than inter-module calls. Finally, a detailed examination of the conversions in the inter-module calls reveals that 70 percent of them use the easy subset of the conversion rules (below the rank of standard conversion). For the rest 30 percent that use standard conversion and user-defined conversion, 24 percent are due to xpcom types. In particular, no user-defined conversion is defined in application modules.

5 Related work

In this section we survey some related work in the areas of empirical study of language or tool use, designing reliable and usable programming languages or APIs.

There are several empirical studies of language or tool usage. Knuth reports a study of the characteristics of Fortran programs in order to understand the effectiveness of compiler optimization and how to improve it [9]. Ernst et al. study the use of the C preprocessor in order to characterize how macros are actually used in practice and the practical implications on tool builders and software developers [5]. English et al. investigate the use of `friend` keyword in practical C++ systems and the appropriateness of such use [4]. Gil and Mamon study the use of micro patterns in large corpus of Java code [7]. Baxter et al. investigate the power law distribution that appear in some structural properties of Java software [1]. Murphy et al. report an analysis of the data they gathered about the usage of the Eclipse environment from actual developers and argue that such data may be used to inform the future evolution of Eclipse [12].

Gannon and Horning describe a study where a set of language features are redesigned with the objective to improve program reliability, which is subsequently empirically verified in terms of both fault frequency and fault persistence [6]. Empirical studies are also conducted to understand the usability tradeoffs of different API design choices, e.g., abstract factory design pattern versus constructors [3] and the usability implications between constructors with and without parameters [15].

In [8], Hoare presents his view on language design. He believes that the goal of a programming language is to assist programmers in the most difficult aspect of programming, that is, on the design, documentation, and debugging of programs, and proposes five objective language design principles to help achieve this goal: simplicity, security, fast translation, efficient object code, and readability. Function overloading contributes to program readability. In [11], Meyer expresses that useful languages cannot be small and proposes as design principles consistency, uniqueness, tolerance and discipline, and methodology. In particular, he advocates that the language designer should provide a methodology for the use of each language feature. There is also much research on designing usable programming languages. A comprehensive survey can be found in [13]. In [2], Cordy describes how to achieve better language usability by paying attention to conciseness, expressiveness, and readability in the design of the Turing language while retaining great power in the notation. He also emphasizes the use of user feedback as a design tool to shape the language design.

6 Conclusion and future work

An empirical study of the use of overloading in Mozilla is described. The goal is to gather evidence on how the feature is actually used, which can be used to inform fu-

ture language design, gather typical use cases, and develop usage guidelines. We conclude that overloading is useful in the systems programming area that C++ is designed for. We find that in Mozilla, the most 'advanced' subset of function overloading are only defined and used in a single utility module xpcom, that the majority of application modules use only the 'easy' subset of function overloading when overloading names, and that most overloaded names are used locally within the modules rather than across module interfaces. We feel that this is potentially a good strategy that can be used to guide the use of overloading in system design. We have also reported an initial set of anecdotes and observations on how overloading is used in Mozilla, from which we hope a useful set of patterns may be distilled eventually by further analyzing data from Mozilla and other systems.

Clearly, our findings are limited only to Mozilla. More systems need to be analyzed before generalization can be attempted. We are currently analyzing other C++ systems. It would also be interesting to perform some in-depth analysis to determine whether overloaded names on module interfaces are necessary or gratuitous. Finally, it may be useful to modify our tool to make the information about overloading available to programmers, for example, by adding an option to the g++ compiler to trigger the output of such information. This can be useful particularly when a programmer needs to understand exactly how an overloading call that involves user-defined conversions is resolved, or when it is desired to systematically inspect all of the narrowing conversions for reliability concerns.

Acknowledgments

The authors are grateful to the three anonymous reviewers for their detailed and constructive feedback that helps improve both the content and the presentation of this paper.

References

[1] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the Shape of Java Software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 397–412, New York, NY, USA, 2006. ACM.

[2] J. R. Cordy. Hints on the Design of User Interface Language Features: Lessons from the Design of Turing. In B. Myers, editor, *Languages for developing user interfaces*, pages 329–340. A. K. Peters, Ltd., Natick, MA, USA, 1992.

[3] B. Ellis, J. Stylos, and B. Myers. The Factory Pattern in API Design: A Usability Evaluation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. English, J. Buckley, and T. Cahill. A Friend in Need is a Friend Indeed. In *International Symposium on Empirical Software Engineering*, pages 469 – 478, Nov 2005.

[5] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002.

[6] J. D. Gannon and J. J. Horning. Language Design for Programming Reliability. *IEEE Trans. Software Eng.*, 1(2):179–191, 1975.

[7] J. Y. Gil and I. Maman. Micro Patterns in Java Code. *SIGPLAN Not.*, 40(10):97–116, 2005.

[8] C. A. R. Hoare. Hints on Programming Language Design. Technical Report STAN_CS-73-403, Standford University, December 1973. In State of the Art Report 20: Computer Systems Reliabilty, C. Bunyan, Ed. Pergamon/Infotech. A 1989 collection of Dr. Hoare's essays entitled Essays in Computing Science published by Prentice Hall, also contains a reprint of this paper.

[9] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practice and Experience*, 1(2):105 – 133, 1971.

[10] S. B. Lippman and J. Lajoie. *C++ Primer*. Addison Wesley, Reading, MA, 1998. Third Edition.

[11] B. Meyer. Principles of Language Design and Evolution. In J. Davies, B. Roscoe, and J. Woodcok, editors, *Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 229–246. Cornerstones of Computing, Palgrave, 2000.

[12] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Softw.*, 23(4):76–83, 2006.

[13] J. F. Pane and B. A. Myers. Usability Issues in the Design of Novice Programming Systems. Technical Report CMU-CS-96-132, Carnegie Mellon University, August 1996.

[14] B. Stroustrup. *The Design and Evolution of C++*. Addison Wesley, Reading, MA, 1998.

[15] J. Stylos and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.