C++ Reflection for High Performance Problem Solving Environments

Tharaka Devadithya[1], **Kenneth Chiu**[2], **Wei Lu**[1]
1. **Computer Science Department, Indiana University**
2. **Department of Computer Science, State University of New York, Binghamton**

Keywords: c++ reflection, problem solving environments, generic programming, scientific computing, high performance computing

Abstract

Problem Solving Environments (PSE) in scientific computing domains require the ability to couple High Performance Computing (HPC) components. A PSE facilitates coupling of tasks or computations in order to aid a scientist in finding a solution to a problem or at least getting closer to a solution. Reflection capabilities are required in order to effectively dynamically couple these components. Reflection facilitates adaptive behavior such as rebinding calls to different functions at run-time, or integrating flexible interpreted languages with compiled languages such as C++ or Fortran. Currently, however, reflection is not available in languages commonly used in high performance computing. While there have been several attempts to incorporate reflection into C++, all of them are either intrusive or are not fully compliant with the C++ standard. In this paper, we present a number of use cases for reflective programming, and show how it can be efficiently and robustly implemented in languages such as C++. Our implementation uses code generation to add metadata, and is fully compliant with the standard C++ specification. We compare the overhead of reflection with languages such as Java, and show that our overhead is acceptable for many scenarios. Our reflection library is open-source, and is available at http://www.extreme.indiana.edu/reflcpp.

1 INTRODUCTION

A Problem Solving Environment (PSE) is a computer system that provides all the computational facilities necessary to solve a target class of problems. These features include advanced solution methods, automatic or semi-automatic selection of solution methods, and ways to easily incorporate novel solution methods [13]. A scientist or engineer should be able to specify a problem symbolically with the notation that they use in communication with each other. Some examples of generic PSEs are symbolic manipulation for basic mathematical expressions, visualization packages for 2-, 3-, and 4-D phenomena, and geometric modelers for a broad class of shapes.

In the scientific computing domain, PSEs would need to couple tasks that are highly CPU intensive. These tasks are generally developed as components in a language used for High Performance Computing (HPC).

A PSE should have a framework that enables a scientist, for example, to dynamically couple tasks or computations, such that the composite will lead to solving some problem. The user interface to couple tasks could be graphical or some kind of a scripting environment. The back-end that supports this capability would need reflection capabilities so that it is possible to dynamically invoke functionalities, which become known only during run-time. Reflection provides a powerful and flexible method invocation ability, as it is even possible to request the application to dynamically load a library and invoke an arbitrary method in it, for example.

Currently, however, reflection is not supported by modern languages used in High Performance Computing (HPC). While there have been attempts to incorporate reflection capabilities to these languages, especially C++[19][5][8][2][12], these approaches are either intrusive[1] or not fully compliant with the C++ standard specification [17]. The intrusive nature is problematic since it would require modifying existing code. Having non-complaint code can cause portability issues as every compiler may not implement non-standard features in the same manner. Both these problems prevent widespread adoption of reflection.

Due to the lack of reflection, PSE developers for HPC domains have to depend on a language such as Java to "wrap" each component with a reflection enabled interface in order to get the maximum dynamic behavior. This approach has two major drawbacks. 1) HPC developers need to be familiar with more that one language or have to depend on other developers to provide reflection capabilities. 2) There is an overhead in transferring data between components developed using different languages, mainly due to the extra copying, since sharing memory between them is not feasible in many cases.

Therefore, there is a strong need for non-intrusive and standard compliant reflection features in C++. In this paper, we show how reflection can be efficiently and robustly implemented in C++, with a complete set of features found in many reflection supporting languages. We compare the overhead of reflection to direct function calls, and with languages such as Java, and show that our overhead is acceptable for many scenarios. We propose our implementation is the only way to incorporate reflection features into C++ in a standard compliant, non-intrusive manner.

2 REFLECTION OVERVIEW

At the code level, there are two types of reflection, namely compile-time and run-time. Compile-time reflection enables accessing type information at compile-time and making appropriate decisions, which facilitates generic programming. An example of such a library that provides some partial information is Boost type traits [1]. Run-time reflection gives the ability to dynamically access type information, such as a list of members in a class, at run-time.

Some programming languages such as Java, C#, and Python support reflection as part of their standard specifications. While C++ supports run time type information (RTTI),

[1]Intrusiveness refers to the requirement of having to add certain annotations to a class so that it is able to provide information about itself.

it only provides very limited features such as testing the type of an expression, querying the type name, etc. The reason for such limited capabilities is that most of the detailed type information is lost during compilation.

Microsoft Visual C++ [7] supports reflection in its managed extensions for C++. However, the resulting applications cannot run natively on an operating system as they need the Common Language Runtime (CLR) [6], and therefore have limited portability.

For high-performance computing, Java and C# have limited applicability, due to performance concerns. For performance, Fortran is still the dominant language, but C and C++ have also made significant strides in this arena. Since C++ is more commonly used for middleware than Fortran, we chose to first develop C++ so that we can more quickly explore its use for adaptivity.

Run-time reflection is implemented in a programming language by including additional metadata into the compiled code. The Java compiler generates this metadata and inserts into the byte code, as the reflection API is part of the Java standard. No fundamental difference between Java and C++ precludes C++ from supporting reflection. Rather, since C++ includes no standard reflection API (beyond RTTI), C++ compilers do not generate the required metadata by default[2]. In order to support reflection, this metadata must be generated as a separate step, and represented as actual C++ code rather than using some lower-level representation.

3 MOTIVATION

Being able to access class information at run time increases the flexibility in programming, since it allows developing generic applications. For example, consider the following simplified code segment.

```
classType = ClassType::getClass(‘‘Service1’’);
obj = classType.createInstance();
obj.invoke(‘‘method1’’);
```

By making *“Service1”* and *“method1”* configurable parameters, this code could be used to invoke any method [3].

As one use case, imagine that we wish to incorporate an existing module into an application. The module has a function named *init_lib()* that must be called before execution. Normally, we might need to write special code to call this function. Using reflection, however, the name *init_lib* can simply be given in a configuration file, and the initialization function can be called dynamically. No code needs to be written, and nothing needs to be recompiled.

Other functions in the existing module can be called similarly. An ontology or other semantic description can be used to match function names and parameters with their meaning, and thus allow automatic gluing between modules [16].

Reflection enables a very powerful mechanism for writing reusable code. The following subsections elaborate on some of the motivations for the need for reflection, especially for developing PSEs.

[2]Some compilers generate debug information that can be leveraged, however.

[3]In this case, any method that does not take any arguments.

3.1 Serialization

Serializing an object state to a byte sequence and deserializing it back is a common approach to communicate between loosely coupled applications. In a PSE the serialized byte sequence can be used to transfer the output from one task to the input of another. With reflection, the sender is able to serialize an object’s state and send it to the receiver so that the receiver is able to dynamically create the objects using their descriptions and populate them with the corresponding state information.

Serialization also enables a generic object persistence framework, since it is just a matter of saving the serialized content to a persistent media.

3.2 Interface to a Scripting Environment

A scripting environment enables a user to invoke commands interactively. However, most high performance application libraries are written in C, C++, or Fortran, which are not directly accessible from a scripting environment. Some scripting languages, such as Python, have the ability to use language bindings (wrapper code that translates back and forth between C/C++ and Python), so that libraries written in C/C++ can be invoked from a scripting environment. However, the binding process still requires a considerable amount of effort and expertise, which a user such as a scientist might not find acceptable.

A reflection-based binding would minimize the effort required for such language binding. In this case, bindings are required only for the reflection related routines, which can easily be automated. PyLCGDict, which is part of the SEAL [9] project, is an example of a module providing reflection-based binding to C++ libraries. It provides a Python interface to the large code base used in many High Energy Physics (HEP) experiments and requires no additional preparations from the Python users. Physicists can then mix and match from a set of component libraries.

3.3 Object Instantiation

The factory method pattern [14] is widely used to polymorphically instantiate objects whose concrete types are not known at compile-time. However, this pattern requires that candidate classes implement a given method such that it creates a new object of the required type. This intrusive nature burdens programmers by requiring to implement such methods for new and already existing classes. Also, the factory method pattern requires all the candidate classes to have a common base class. An approach using reflection would not have such restrictions and will not be intrusive.

4 GOALS

We intend the C++ reflection library to provide features that other reflection-supporting programming languages offer as much as possible. Our main goals are listed below:

Run-time access to members. It should provide run-time access to member functions and data members[4] in a class. This mainly boils down to being able to invoke member functions and read from / write to data members.

[4]Member function and data member are the terms used in the C++ specification.

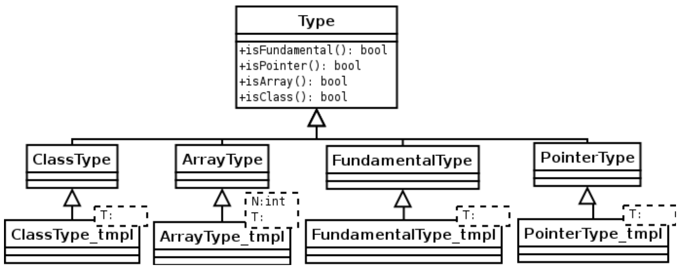

Figure 1: Metaclass hierarchy.

Relatively complete. At a minimum, it should provide information about classes and their members and the ability to access them.

Standard compliant. The reflection library should be compliant with the ISO C++ standard [17]. For example, the standard does not guarantee that static offsets from an object's address would point to a given data member in that object, except for "plain old data" (POD) types, which essentially correspond to C-style structs. Being compliant increases the portability of the library.

Relatively efficient. While accessing classes and their members via reflection incurs some overhead, the resulting code that uses reflection should be relatively efficient compared to a code that does not. We intend to minimize the overhead by using C++ templates to record most of the required metadata at compile time.

Non-intrusive. We intend to incorporate reflection without requiring any modifications to existing code. Instrumentation will make it difficult to use reflection with already existing code. Moreover, it creates the possibility of programmers making errors in meta-information related sections.

5 C++ REFLECTION LIBRARY

As discussed in Section 8, other implementations of C++ reflection achieve some of the goals described in the previous section. However, they are either intrusive or violate the C++ standard to a certain extent. We thus developed our library with non-intrusiveness and standard compliance in mind.

Our library consists of a small set of core classes, and another set of generated classes to store and present the type-specific, metadata information. The type-specific, metadata classes are generated by parsing the user-supplied target classes, and then traversing the syntax trees to generate the metadata classes.

In this section, we describe the library from a user's perspective.

5.1 Metaclasses

ClassType is the metaclass used to represent user defined classes. These classes are derived from a more generic class, *Type* as shown in Figure 1.

Class information can be obtained as follows using *ClassType*.

```
ClassType *ct =
    ClassType::getClass("ClassA");
string className = ct->name();
```

In this example, *ClassA* is the target class about which *ct* maintains information.

ClassType, along with its helper classes, provides the following features.

- Name and type
- List of data members and member functions
- Instantiation interface
- Inheritance information

Once an instance of *ClassType* is created, the actual target object (i.e., an instance of *ClassA*) can be created as,

```
classAObj = ct->createInstance();
```

The other derived types, *FundamentalType*, *ArrayType* and *PointerType* as their names imply are used to represent primitive types (e.g., integer, double), arrays and pointers, respectively. While these types do not have most of the interfaces related to user-defined classes, they provide a uniform way to access the basic information about a variable such as its name and type.

5.2 Member Classes

There are two types of member classes, namely data members and member functions. These encapsulate the members of a class and provide means of indirectly accessing them, given their names at run-time. Instead of maintaining the offset to the encapsulated member, each member class maintains a pointer to its respective data member or member function. This ensures that this library is fully compliant with the C++ standard.

5.2.1 Data Members

Each data member in a class is represented by a *DataMember* object. This object maintains information such as name and type. The member function, *ref()* returns a reference to the represented object, allowing the user program to manipulate it as if it had direct access to the data member. The data member can be accessed as:

```
DataMember dm = ct->getDataMember("f1");
dm.ref<int>(&classAObj) = 1234;

BoundDataMember bdm =
  ct->getBoundDataMember(&classAObj, "f1");
// bdm is completely generic.
bdm.ref<int>() = 1234;
```

BoundDataMember, is aware of the associated object and therefore it is not required to pass a reference to *classAObj*, as with *DataMember*. A significant characteristic of bound data members is that they are now completely type-independent of the underlying object. They can thus be passed around to code that is completely generic.

5.2.2 Member Functions

Analogous to data members, member functions are represented by *MemberFunction* classes. In addition to the name and return type of a member function, *MemberFunction* provides the following interfaces.

Arguments. A list of the arguments is maintained as a vector of generic *Type* (refer to Section 5.1) objects. This enables the argument types to also be fundamental types, pointers, or arrays in addition to user defined classes.

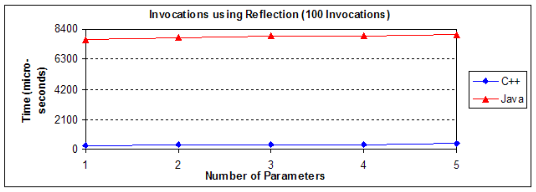

Figure 2: Invocation times using Reflection.

Invoke. Two main types of interfaces are provided for invoking a member function. The first type requires the arguments to be passed individually along with a pointer to an instance of the corresponding class. This type has several overloaded *invoke()* methods, each taking a different number of arguments.

Following is an example of invoking a member function.

```
MemberFunction mf = ct->getMemberFunction("m1");
mf.invoke<int>(&classAObj);
```

The return type of *m1* is *int* and the member function does not take any arguments.

The second type takes the arguments as a vector and therefore needs only a single interface to be defined for each type.

The analogous class to *BoundDataMember* for data members is *BoundMemberFunction*. Such an instance can be passed to generic code that is type-independent of the underlying types, which is an important characteristic for building generic adapters.

6 PERFORMANCE MEASUREMENTS

Reflection provides a level of indirection and therefore incurs a performance penalty. We conducted some performance measurements in order to find out this overhead in the C++ reflection library. We also conducted similar tests on Java reflection. We, however, did not conduct any comparisons against the other C++ reflection approaches, mainly since our objective is not to present a faster reflection approach here. As noted earlier, we are presenting a reflection library that is compliant with the C++ standard and is not intrusive, which are features that are not found in other C++ reflection approaches.

Each test involved dynamically instantiating an object and invoking a method in it. Tests were conducted with five such methods having 1, 2, 3, 4 and 5 arguments, respectively. Each time, methods were invoked as direct and virtual function calls and also using reflection. In the case of using reflection, the object also was instantiated using reflective methods. We invoked each function 100 and 100,000 times, respectively. Figure 2 shows the times taken for 100 invocations using reflection in C++ and Java.

We were more interested in finding out the overhead of using reflection as opposed to the time for reflection calls. We calculated this overhead as the extra time taken by using reflection as a percentage of the time for a direct call. The timings for virtual functions did not show any noticeable difference from that of direct function calls, probably because there were not many entries in the virtual table, and therefore were not considered for the overhead calculations.

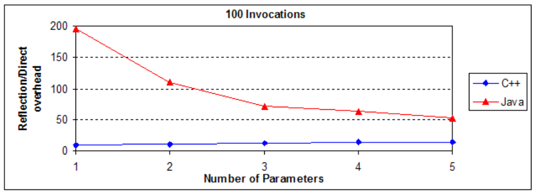

Figure 3: The overhead of using reflection for 100 invocations of the same function.

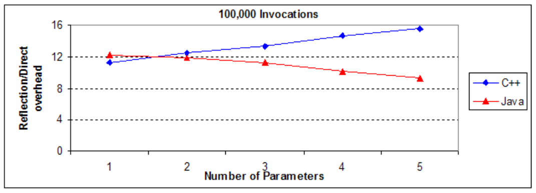

Figure 4: The overhead of using reflection for 100,000 invocations of the same function.

The tests were conducted on a Dell GX620 3.2 GHz machine. It has 1 GB of RAM and is running Gentoo Linux. The C++ version was compiled with g++ 3.3.6 with -O3 optimization, while the Java version was compiled and run with JDK 5.0 update 3 with *-server* option.

6.1 Results and Analysis

Figures 3 and 4 show the performance measurements for 100 and 100,000 invocations, respectively.

It can be observed that the overhead for C++ reflection increases with the number of arguments. This is mainly due to constructing the argument list dynamically, which involves few small amounts of memory allocations. For Java reflection, the overhead decreases with the number of arguments. However, this was because the time for direct calls in Java increases significantly with the number of arguments, resulting in a decreasing overhead percentage. The time for direct calls in C++ does not increase in a significant manner with the number of arguments.

Figure 3 indicates that for a small number of invocations, such as 100, the overhead of C++ reflection is less than that for Java reflection. However, for a large number of invocations, the overhead of Java reflection is less when there is more than one parameter, as seen in figure 4. The main reason could be the fact that the Java compiler optimizing the reflection related part of the code, after noticing the high frequency of usage.

A profiling analysis of the C++ version using gprof [15] showed that around 60% of the time is spent in member function lookup. Currently a map in the C++ standard library is used as the container for member function objects of a class. The map is implemented as a balanced binary tree, which has a time complexity of $O(\log n)$, in the number of names. Furthermore, balanced binary trees must perform multiple string comparisons. Performance could be improved by using a hash table or a trie, instead. Both hash table (with perfect hashing) and trie have time complexity of $O(1)$.

7 APPLICATIONS

In this section, we examine how C++ reflection can be applied to a number of use cases.

7.1 Remote Method Invocation

We concentrate on the ability to invoke a method at run time, given the class and method names along with the arguments. While we experiment with a remote method invocation in this section, it should be noted that a PSE will also require a similar type of invocation, given the method name and parameters.

7.1.1 Code Fragment

Consider the following XML fragment with the invocation information, which can be considered as a simplified *Body* of a SOAP [20] message.

```
<chargeReservation>
  <reservation>
    <code>FT35ZBQ</code>
  </reservation>
  <creditCard>
    <name>Eke Jsgvan Xyvind</name>
    <number>123456789099999</number>
    <expiration>
      <year>2005</year>
      <month>2</month>
    </expiration>
  </creditCard>
</chargeReservation>
```

The following code performs the invocation after reading the XML.

```
XMLDocument dom;
dom.parse(...);

classType =
    ClassType::getClass("Service");
obj = classType.createInstance();
operationName = dom.getRoot()->getName();
memberFunction =
  classType.getMemberFunction(operationName);
Arguments args;
extractArgs(memberFunction, dom.getRoot(), args);
memberFunction.genericInvoke<void>(*obj, args);
```

Apart from the hard-coded class name of *Service*, this code can be reused to perform any invocation, given an XML document in the correct format at run-time.

7.1.2 RMI Performance Results

We conducted some tests, in order to measure the performance drop of using reflection relative to using generated code. For both tests, we used 5 XML messages, each containing the name of the method to be invoked and its arguments. The messages had methods of 1, 2, 3, 4 and 5 arguments, respectively. Both versions invoked the same methods, which performed a simple string addition.

Figure 5 shows the performance measurements of the two versions for 100 invocations of the same method. On average, the reflection version is 2.5 times slower than the generated code. However, we expect that most of the time, reflection is used for loose-coupling, and so invocation costs would be amortized over a significant amount of computation. Also, the flexibility obtained by using reflection as opposed to generated code outweighs this small overhead.

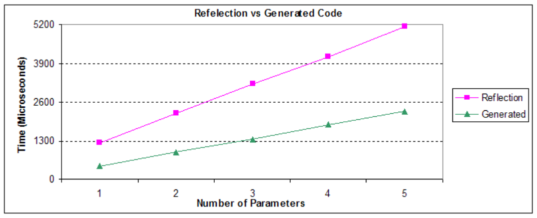

Figure 5: Using reflection vs. generated code for remote method invocation (100 invocations).

7.2 Solving Large Sparse Linear Systems of Equations

Solving sparse linear systems is a task that involves a significant amount of experimenting, since no current mathematical theory provides a practical guide to choosing a solution strategy. A linear algebraist might need to try out different strategies before even coming closer to a solution.

A PSE could help the algebraist by providing a set of components out of which he or she can choose the ones that provide the functionalities required by the current strategy. An example of such a PSE is The Linear System Analyzer (LSA) [11]. Examples of the components of LSA are,

- I/O components for getting systems into and solutions out of the LSA
- Filter components for manipulating the systems with reorderings scalings or dropping of entries based on their relative sizes
- Solver components for actually solving the systems
- Information components for providing some analysis of the systems

A PSE developed using reflection would enable dynamic coupling of these components. Without reflection, the PSE would need to know of all the components that can be used. Any new components will require modification to PSE code. Also, PSE code will have to statically distinguish between types with a switch statement block.

Some use of component descriptors might alleviate some of these restrictions, but then these would essentially become an ad hoc kind of reflection system. By implementing reflection once, in a generic and flexible manner, the benefits can be extended to many different types of PSE-like systems.

7.3 Unit Testing Framework

Currently available unit testing frameworks for C++, such as CppUnit [3], uses function pointers to access member functions that are to be treated as the test functions. The test code developer needs to explicitly “register” these member functions so that the “Test Runner” can invoke them during a test run.

Reflection provides a more convenient approach to develop a unit testing framework, which does not require such explicit registration of each member function. Instead, the test writer could prefix each such member function name with some meaningful word, (such as “test”) and inform the “Test Runner” to execute all member functions whose names have that prefix.

8 RELATED WORK

We present some of the related work in supporting reflection in programming languages.

SEAL Reflex [19] is a multi-platform C++ reflection library, with no external dependencies. The code is "almost" compliant with the C++ standards. It is part of the SEAL core libraries and services project [9]. The goal of the SEAL project is to provide the software infrastructure, basic frameworks, libraries and tools that are commonly used in the Large Hadron Collider (LHC) [10] Computing Grid (LCG) Project [4].

SEAL Reflex library uses offsets from the object pointer to access data members. This is not guaranteed by the C++ standard [17] to work for non-POD types, which may be an issue for a broader application of reflective middleware approaches.

Metaclasses and Reflection in C++ [5] is based on **Meta Object Protocol (MOP)** [18]. The idea of MOP is to make the class definitions normal objects where the object properties are normal attribute values of the class definitions that can be manipulated at runtime. Since C++ does not have such features, MOP needs to be provided by a library. However, the approach taken involves modifying existing classes, to some extent.

Reflection for C++ [8] uses two approaches for supporting reflection. The first extracts debug information from the binary. This requires that the program be compiled in debug mode and therefore not suitable for production level code. The second approach requires the programmer to provide meta-information as a set of macros within the classes themselves, which would involve instrumenting existing classes.

C++ Reflection [2] by Fabio Lombardelli provides full reflection for C++ through template metaprogramming techniques. While it is fully compliant with the C++ standards, it requires the programmer to annotate the classes in order for them to be reflective.

Bjarne Stroustrup, in 2001, had proposed a native extension of C++ with reflection information, called **eXtended Type Information** (XTI). While there has not been much development recently on this topic, it is very likely that the next version of C++ (i.e., C++0x) will have support for reflection. However, the next C++ standard is not scheduled to be released soon.

Chuang et al. presented a reflection implementation **based on meta objects** [12]. It defines a separate meta object for each class, that completely captures information of the class for introspection purposes. However, it maintains static offsets to class members. Furthermore, it is intrusive as it inserts friend functions into the candidate classes, in order to gain offset information about private members.

9 CONCLUSION

Problem solving environments are becoming popular among scientists for experimenting and solving problems by dynamically coupling different tasks. The support for reflection is an essential feature in order to get the maximum benefit of dynamic coupling. Currently reflection is not available in languages used in high performance computing.

In this paper, we have shown how reflection can be added to C++, a language which is gaining acceptance for high-performance computing. Our approach should be applicable to other HPC languages, also. Our implementation's efficiency compares favorably to that of Java. We have shown that the overhead due to reflection is not very high.

The main objective of our experiments were to show that the overhead of reflection in C++ is comparable to that in languages that do support reflection. Also, our intention was not to present a better, faster way to do reflection in C++. What we presented was what we consider to be a good way of doing reflection, and so far it's the only known way to do reflection in C++ that is standards compliant and non-intrusive.

References

[1] Boost Type Traits. `http://www.boost.org/doc/html/boost_typetraits.html`.

[2] C++ Reflection. `http://sourceforge.net/projects/cppreflect`.

[3] CppUnit - A C++ Unit Testing Framework. `http://cppunit.sourceforge.net`.

[4] LHC Computing Grid (LCG) Project. `http://lcg.web.cern.ch/LCG/`.

[5] Metaclasses and Reflection in C++. `http://www.vollmann.com/pubs/meta/meta/meta.html`.

[6] Microsoft .NET Framework. `http://msdn.microsoft.com/netframework`.

[7] Microsoft Visual C++. `http://msdn.microsoft.com/visualc`.

[8] Reflection for C++. `http://www.garret.ru/~knizhnik/cppreflection/docs/reflect.html`.

[9] SEAL - Core Libraries and Services Project. `http://seal.web.cern.ch/seal/`.

[10] The Large Hadron Collider (LHC). `http://public.web.cern.ch/public/Content/Chapters/AboutCERN/CERNFuture/WhatLHC/WhatLHC-en.html`.

[11] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, E. Akman, J. Balasubramanian, F. Breg, S. Diwan, and M. Govindaraju. The linear system analyzer. *Technical Report TR-511, Computer Science Dept, Indiana University*, 1998.

[12] T. R. Chuang, Y. S. Kuo, and C.-M. Wang. Non-intrusive Object Introspection in C++: Architecture and Application. In *International Conference on Software Engineering*, 1998.

[13] E. Gallopoulos, E. Houstis, and J. R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Comput. Sci. Eng.*, 1(2):11–23, 1994.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[15] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, 1982.

[16] A. Gupta et al. Registering Scientific Information Sources for Semantic Mediation. In *21st International Conference on Conceptual Modeling*, 2002.

[17] ISO. INTERNATIONAL STANDARD: Programming languages - C++. `http://www.open-std.org/jtc1/sc22/wg21`, 1998.

[18] G. Kiczales. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[19] S. Roiser. The SEAL C++ Reflection System. In *Computing in High Energy and Nuclear Physics (CHEP)*, September 2004.

[20] W3C. Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`, 2000.