Reification and Reflection in C++:

An Operating Systems Perspective*

Peter W. Madany, Nayeem Islam,

Panos Kougiouris, and Roy H. Campbell

University of Illinois at Urbana-Champaign

Department of Computer Science

1304 W. Springfield Avenue,

Urbana, IL 61801

1 Introduction

In this paper we will examine the benefits of applying the concepts of reification and reflection[12] to C++[13] in order to support the construction of an object-oriented operating system. We describe ways in which C++ programs can perform reflective computation and to what extent the language and the run-time environment support this computation. To evaluate the benefits of reflection within C++, we report on the effect of reflective facilities on the design of *Choices*[1, 10, 11], a framework we are constructing for object-oriented operating systems. Our paper describes *practical* reflective computation in a real system and represents an application in C++ of ideas discussed by Maes[8].

Reflective facilities can be built into the architecture of an object-oriented language as proposed by Maes. Alternatively, researchers have proposed using a multi-level system architecture containing meta-

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grant NAG 1-163.

objects[16, 15]. However, we adopt a more conservative approach in which we build selected reflective facilities within C++, an existing programming language architecture, enhancing the run-time system, and using preprocessors and programmer conventions. The resulting facilities simplified the implementation of many *Choices* operating system functions.

In Section 2 we refer to other work on reflection and define the terms that we use in the later sections. In Section 3 we present the language aspects on which the *Choices* object-oriented operating system reflects, and we discuss the techniques and conventions that we used to enable this reflection. In Section 4 we conclude by evaluating the effort needed to add reflective facilities to C++.

In the full paper we will elaborate on the discussions presented in this extended abstract, and provide details of the reflective facilities that we have constructed and used.

2 Definitions

Reification and reflection are terms that Brian Smith[12] introduced in his PhD thesis concerning the structuring and organizing of self-modifying procedures and functions. Pattie Maes[8] broadened the application of these ideas and commented that "the concept of reflection fits more naturally in the spirit of the object-oriented world." Recently, other researchers have elaborated on reflection in object-oriented systems and have presented examples of how an existing language can be modified to support reflective facilities.[3, 4, 14]

In the context of an object-oriented programming language, we define *reflection*[1] as the ability of an executing system of programmed objects to make attributes like the invocation, interface, inheritance and implementation of the objects to be themselves the subject of their own computation. Although we agree with the definitions given by the authors mentioned above, we view reflection from a bias of building an operating system that supports object-oriented programs. The steps involved in reflection consist of: *reification* of an abstract object-oriented concept, *reflective computation* using the reified attribute as data, and *reflective update* that modifies the objects through reflective computation.

[1] The Oxford English Dictionary[9] defines reflection as "the mode, operation, or faculty by which the mind has knowledge of itself and its operations, or by which it deals with ideas received from sensation and perception."

We define *reification*[2] as the representation of an attribute of an object-oriented program such as member function dispatch, inheritance, and object structure as an object within that program. Such a reifying object provides member functions that support reflective computation or reflective update.

In our C++ object-*oriented* environment, a reifying object is defined by a concrete reifying class. Instantiation of such a class constitutes reification of the attribute. The concrete reifying class inherits properties defined by a class hierarchy. In particular, it may inherit properties from an abstract reifying class that defines an abstract attribute. The dispatch of a member function of the reifying object corresponds to reflective computation. If the reflective computation updates the reified data, this corresponds to reflective update. The reifying object has a *causal connection* with its attribute[8]. The dispatch of a member function performing a reflective update will modify the attribute. Similarly, a change of the attribute modifies the reifying object.

The most important decisions to be made in a reflective system are what attributes of the system should be considered reifiable. Common examples of reified attributes in a C++ object-oriented application include object member names, object data members, object member functions, and the member function dispatch mechanism. However, to avoid the dangers associated with self-modifying instructions, it is not prudent to allow reification of instruction code. Depending on the application, however, several other attributes could also be reified, for example the protection afforded the encapsulated data of an object. Further, a reified attribute could be associated with an object or a class of objects.

3 Examples of Reflection

As part of the *Choices* project[1], we have developed an object-oriented operating system written in C++. Reflection in *Choices* supports:

- Storage — concurrency and performance of memory allocation and deallocation.
- Existence — mechanisms for automating the deletion of objects.

[2] Reification [9] is "the mental conversion of a [person or] abstract concept into a thing."

- Persistence — mechanisms for persistent object activation and deactivation.
- Class — the relationship between an object and its class.
- Inheritance — the relationship between classes in a class hierarchy.
- Encapsulation — hardware-enforced encapsulation of instance data.
- Inspection — human-readable representations of objects for tracing, debugging, and instrumentation.

In the following paragraphs we discuss the motivation for each category of reflective computation, the attributes that are reified, the reifying abstract classes, the member functions that define the reflective computation, and the concrete classes that implement these member functions. We also discuss how we use C++, additional tools, and programmer conventions.

Storage The memory for C++ objects in *Choices* is allocated from heap storage. While *Choices* boots, few operating system facilities are available. Therefore the initial heap manager uses a simple algorithm that has few features and places few requirements on the operating system. As the boot progresses and both virtual memory and process-switching facilities become available, the default heap manager is changed to a multi-threaded allocator that provides an appropriate balance of space and time usage properties for a multi-threaded kernel. The C++ language directly supports reflection on heap management by allowing the programmer to overload the `new` and `delete` operators for both the entire system and for individual classes.

A reifiable heap manager has proved essential for building an operating system. In *Choices* an Allocator object reifies the heap manager. The Allocator class is an abstract reifying class that defines `allocate` and `free` functions, which are invoked by the C++ operators `new` and `delete`. The heap can be replaced in a running system by calling the global `SetNewAllocator` function with the address of the new Allocator object. Several concrete subclasses of Allocator implement reified heaps with various properties; examples include SimpleAllocator and ConcurrentAllocator.

Existence Several object-oriented languages, including Smalltalk[6], implicitly delete objects when they are no longer of use. In C++, however, objects must be deleted explicitly. This is justified because in all but the most trivial object-oriented systems, it is impossible to determine at compile-time when an object should be deleted, and C++ is designed to avoid the overhead of a built-in garbage collector. In *Choices*, however, the dynamic use of objects in the system would impose a significant burden on the programmer unless implicit deletion is supported. Both reference-counting or garbage collection techniques can support implicit deletion; we have chosen to use reference-counting.

Reference-counting implies that an object should exist as long as there is at least one pointer to the object. In *Choices* the number of pointers to an object is reified by the concrete reifying class ReferenceCount. Class Object defines three member functions, `reference`, `unreference` and `noRemainingReferences`,[3] that perform reflective computation by incrementing and decrementing an Object's ReferenceCount and calling its destructor when the ReferenceCount is decremented to zero.

The mere provision of reference-counting functions does not ensure that ReferenceCounts will accurately reflect the state of the system, and even the slightest inaccuracy can corrupt a running system. Therefore, we chose to reify not only the reference-count for each Object but also pointers to objects, using a reifying class called ObjectStar. ObjectStars automatically call `reference` when assigned the address of an object, and they automatically call `unreference` when the address they store is overwritten. By ensuring the accurate usage of reference-counting functions, ObjectStars effectively implement the desired control over an object's existence.

Because pointers are statically-typed in C++, we use a shadow hierarchy of ObjectStars, which mirrors the class hierarchy in *Choices*. The shadow hierarchy allows usage of ObjectStar subclasses to be type-checked at compile-time. C++ does not directly support implicit deletion of objects, therefore we developed a tool to create the ObjectStar shadow hierarchy. We also had to agree on a programmer convention that ObjectStars would be used instead of traditional C++ pointers.

[3] These functions are inherited by all subclasses of Object.

Persistence Most operating systems support the storage of persistent data on secondary memory devices, usually this support takes the form of a file system. In some object-oriented systems including *Choices*, a persistent object may be used to encapsulate persistent data. The persistent object simplifies storing and retrieving persistent data by making those operations part of a programmer transparent object activation and deactivation scheme. When no processes are accessing a persistent object, it is automatically stored on secondary storage. When a process references a persistent object, it is automatically retrieved. A persistent object has a global lifetime that is independent of the process that creates it. It persists as long as there is at least one reference to it from another persistent object. Applications can dynamically identify a set of "root" objects that are by definition persistent, regardless of whether any other object refers to them.

As with existence, either reference-counting or garbage collection can control persistence. Our first implementation of persistence uses reference-counting; the number of persistent references to a persistent object is reified by the concrete reifying class LinkCount[4]. A PersistentObject has a LinkCount object and `link` and `unlink` member functions that perform reflective computation on that object. When the ReferenceCount of a PersistentObject is decremented to zero, the PersistentObject is written to secondary storage. When the LinkCount of a PersistentObject is decremented to zero, the PersistentObject is removed from secondary storage. The PersistentObjectStar class and its subclasses define pointers to instances of PersistentObject and its subclasses. PersistentObjectStars not only call `reference` and `unreference` functions when appropriate but also call `link` and `unlink` functions of PersistentObjects.

If *Choices* used garbage collection to control persistent storage, then the structure of each subclass of PersistentObject would need to be reified. A garbage collector would use the reified structure of PersistentObjects to follow references between PersistentObjects and to determine the PersistentObjects that are reachable from the set of root objects, and it would delete unreachable objects. C++ supports neither reference-counted nor garbage-collected persistent storage. As with controlling object existence, we developed a tool to create the PersistentObjectStar shadow hierarchy, and we had to rely on programmer

[4] We named the persistent reference-count "LinkCount" because of its similarity to the number of links in a UNIX file system.

conventions.

Class The presence of classes as objects at run-time can simplify software development, enable dynamic extensions, and enhance debugging facilities.

Class is a concrete reifying class that reifies C++ classes as objects in *Choices*. Classes are similar to the Dossiers described in [7], but extended to support dynamic code linking and portable debugging. At run-time, every Object in the *Choices* system refers to a specific Class. These Objects have the reflective computation functions `isMemberOf` and `isKindOf` to test whether they belong to particular classes or hierarchies. Class objects are also inserted in the *Choices* NameServer, this allows user programs to access Classes dynamically by name.

C++ does not directly support classes as objects, therefore we developed a tool to create the code that instantiates Classes.

Inheritance Since *Choices* uses inheritance extensively, merely knowing whether an object belongs to a class is insufficient to support dynamically extensible systems, the entire static class hierarchy and inheritance relationships must represented at run-time. The concrete reifying class Class reifies the class hierarchy used to build the *Choices* operating system.

In C++ the member functions of an object can be dynamically bound using the virtual function table. However, constructors, which assign virtual function table pointers to objects, cannot be dynamically bound. Therefore, the classes of objects must be fully defined at compile-time, and objects' member functions must also be fully defined at compile-time. This implies that, once a system has been compiled, the classes of its objects and their member functions are fixed.

Choices is designed as a run-time extensible system. Constructors are reified as the `constructor` member of class Class. This allows programs to specify objects as instances of abstract classes whose concrete subclass implementations can be changed at run-time.

Classes representing concrete subclasses can be added to a running system, and if the code for a concrete subclass is not present at run-time, *Choices* uses an instance of the abstract reifying class CodeLoader that

reifies the symbol table. A CodeLoader object locates the code in the file system, loads the code into memory, resolves the undefined symbols in the loaded code, and installs loaded constructor and member functions. The CodeLoader is similar to the dynamic addition of code described in [2]. The CodeLoader maintains the causal connection between the reified hierarchy and the actual hierarchy by loading class definitions dynamically. It provides the member functions `resolveUndefinedSymbols`, `installSymbols`, `relocateSymbols`, to perform reflective computation. A concrete reifying subclass, COFFLoader, implements the functionality of the abstract class.

We found that an entire shadow hierarchy of Classes was unnecessary, since Classes can refer to their superclasses and subclasses. Again, C++ directly supports neither a dynamic class hierarchy at run-time nor the addition of code at run-time, therefore we developed the Class and CodeLoader classes.

Encapsulation C++ provides data encapsulation facilities, including *public*, *protected*, and *private* member data and functions. These protection facilities are not hardware-enforced and thus can be compromised at run-time. Hardware-enforced encapsulation is essential to an operating system. This encapsulation can be achieved using separate address spaces, virtual memory restrictions, and the supervisor state of processors.

Normally objects cannot invoke functions across such hardware protection boundaries. To overcome this restriction, instances of the concrete reifying ObjectProxy class[10] reify run-time encapsulation of system objects. An ObjectProxy represents an object that is encapsulated in a different hardware-protection region. Invocations of ObjectProxy member functions lead to an invocation of the reflective computation member function `delegate` that crosses the appropriate protection boundary and delegates the member function invocation to the real system object. An ObjectProxy cannot be forged, therefore data encapsulation can be guaranteed.

Inspection When inspecting objects for debugging and displaying objects for tracing, much more than the object's address is needed. Furthermore, different classes of objects will require different kinds of inspection.

The concrete reifying class Object reifies human readable information about an Object. This class defines the reflective computation member functions `writeName` and `inspect` that display the Class, address, and an

optional symbolic name to identify an object when debugging the system. Subclasses of Object can redefine the functions to provide additional helpful information. The `setName` member function of class Object allows objects to change their symbolic name when necessary.

While one cannot consider the symbolic name of an object a reification of its identity, it is a reification of how an object identifies itself to a user of the system. If the structure that C++ classes impose on objects were reified,[5] that reified structure could be used to further enhance object display.

4 Conclusion

In the final paper we will give further details and more examples of how reflective facilities can help one build an operating system in an object-oriented language like C++. Although C++ does not provide specific support for reflection, facilities can be added by extending the C++ run-time system, developing C++ preprocessors, and using programmer conventions.

Our approach has several disadvantages. While C++ is flexible and allows reflective facilities to be built, the facilities are fragile and must be used carefully. For example, our preprocessors do not fully parse and analyze a C++ program, and incorrect use of the reflective extensions may generate incorrect results without a compile time warning. Programmers invariably break any conventions that are not enforced by a compiler. Furthermore, extensions to the C++ run-time system in *Choices* may be difficult to replicate on other operating systems.

While modifying C++ to include a reflective architecture might be a laudable but impractical goal, there are smaller changes that could be made that would better support the expression of reflection. C++ could provide run-time representations of: the class of an object, the structure of instances of a class, and the superclasses and subclasses of a class. This would simplify and make more robust the reification of classes, inspection, and inheritance. The C++ language could also support the overloading of pointer creation, assignment, and deletion. This would obviate shadow hierarchies of pointers to various classes of objects.

[5] See [5] for a macro-preprocessor implementation of reified class structures.

We **do not advocate** modifying C++ to support the various reflective facilities discussed in this paper. These features seem to require fundamental changes to the architecture of the language. However, language designers that are basing their language designs upon the success of C++ should consider reflective facilities.

References

[1] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of Object-Oriented Operating System Design. Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[2] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, San Francisco, California, April 1990.

[3] Jacques Ferber. Computational Reflection in Class based Object Oriented Languages. In *OOPSLA 89, Conference Proceedings*, pages 317–326. ACM, 1989.

[4] Brian Foote and Ralph E. Johnson. Reflective Facilities in Smalltalk-80. In *OOPSLA 89, Conference Proceedings*, pages 327–335. ACM, 1989.

[5] Erich Gamma, André Weinand, and Rudolf Marty. Integration of a Programming Environment into ET++: A Case Study. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 283–298, Nottingham, UK, July 1989. Cambridge University Press.

[6] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[7] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[8] Pattie Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA 87, Conference Proceedings*, pages 147–155. ACM, 1987.

[9] Oxford University Press. *Oxford English Dictionary.* Oxford University Press, Walton Street, Oxford, UK, 1971.

[10] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[11] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[12] Brian C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, January 1982.

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[14] Takuo Watanabe and Akinori Yonesawa. Reflection in an Object-Oriented Concurrent Language. *ABCL An Object Oriented System*, pages 45–70, 1990.

[15] Yasuhiko Yokote, Fumino Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The muse object architecture: A new operating system structuring concept. Technical Report SCSL-TR-90-012, Sony Computer Science Laboratory Inc., October 1990.

[16] Yasuhiko Yokote, Fumino Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 89–108, Nottingham, UK, July 1989. Cambridge University Press.