The Iguana Experience: Meta-Level Programming in a Compiled Reflective Language

Implementing Object Persistence

Peter Haraszti, Tilman Schäfer, and Vinny Cahill

Distributed Systems Group
Department of Computer Science
Trinity College Dublin, Ireland
{peter.haraszti,tilman.schaefer,vinny.cahill}@cs.tcd.ie

Abstract. Reflective programming languages allow the underlying object model of the host language to be extended with features such as persistence, remote method invocation, fault-tolerance and transactions while providing a strong separation of concerns and transparency. However, existing architectures, especially for compiled languages such as C++, have been found to be too restrictive in that the binding of objects with a particular object model is static, resulting in less efficient implementations. In this paper we describe our experiences with implementing persistent objects using the Iguana reflective programming language.

1 Introduction

Reflective programming languages have been used in a number of case studies in order to provide applications with extended object behaviour, for example object persistence [KYK+99], fault-tolerance [KFRGC98] and atomicity [SW95]. Although it has been shown that reflection offers some advantages over traditional approaches, including a strong separation of concerns and transparency, it cannot be denied that meta-level programming remains a rather neglected programming paradigm that has not yet found its way into widespread use. We therefore took a well researched example of extended object behaviour, persistence, and compared a representative, non-reflective implementation (Texas/C++ [SKW92]) with an implementation using the Iguana reflective programming model [GC96]. This implementation work has given us some exposure to meta-level programming. We were specifically interested in using a compiled language such as C++, mainly because interpreted languages may not be appropriate for low-level system programming. Moreover, reflective extensions of compiled languages have previously been found to be too restrictive in that the binding of objects with a particular meta-level representation can usually occur only once, a shortcoming that we tried to address in Iguana. In this paper, we describe our experiences with reflective programming in Iguana/C++.

2 Overview of Iguana

Iguana [GC96] [Gow97] is a run-time meta-level architecture similar to OpenC++ v1 [Chi95] [Chi93]: base-level objects are associated with metaobjects, each of which representing or implementing a specific language construct. Programmers can substitute the default semantics of each of the language constructs individually by providing customized implementations.

A design issue that distinguishes Iguana from comparable platforms is that it is dynamic in nature: Iguana offers the dynamic and selective reification of language constructs at run-time, meaning that customised object models can evolve as the system runs. We have shown that this is particularly important for building systems with a strong need for dynamic adaptation, both at design time (software evolution) and run-time (dynamic reconfiguration), as in middleware and operating systems [DSC+00].

The Iguana model provides the concept of a *protocol* both as a means of defining a new metaobject protocol (MOP) and of specifying the implementation of a MOP. In Iguana, the definition of a MOP specifies the set of language constructs to be reified by objects that select the MOP as well as the set of metaobject classes to be used to implement those reified constructs. The language constructs (called reification categories) are: Class, Attribute, Method, Creation, Deletion, Invocation, StateRead and

StateWrite. These metaobject classes extend the default classes in order to alter the default behaviour in the desired way. Metaobjects can be either *shared* between all instances of a class or *local* to a particular object. Classes or individual objects can be associated with protocol definitions by means of the protocol selection operator (==>). See Section 3 for examples of protocol definition and selection.

Iguana supports multiple inheritance of protocols: a protocol may extend other protocol(s). The set of active reification categories for a given object can increase or decrease over time by means of dynamic protocol selection.

3 Implementing persistent objects using Iguana/C++

Object persistence is implemented as a set of Iguana/C++ protocols based on the Tigger object support framework [Cah99]. Any type can potentially persist, provided they are associated with the Persistent protocol. Pointer semantics between persistent objects are preserved. Classes can include primitive types, class, pointer and array attributes. The broad range of C++ language issues arising when designing a persistent C++ extension can be found in [KYK+99].

In our design, we adopted the persistence by reachability approach to determine which objects are to be retained. In other words, potentially persistent objects that are transitively reachable from a distinguished persistent root via references will persist between program executions. The states of these objects are stored in an underlying Persistent Object Store (POS). The implementation of the Persistent protocol interfaces to the underlying POS and initiates the loading and storing of objects in the POS as necessary as well as detecting accesses to non-resident objects.

While the use of reflection is essentially transparent to the programmer, the use of persistence is not. This means that the application programmer must be aware of the specifics of the model of persistence provided by the Persistent protocol, the implications of persistence by reachability and the physical location of the POS files. When working with persistent objects, the application programmer may need to distinguish between the cases where a new object needs to be created and initialised for the first time versus the case where the object has been created by a previous execution of the program. To allow programs to refer to previously created objects, a simple name service is provided, which allows the association of symbolic names with persistent objects.

The programming interface to the meta-level is designed in form of an Iguana/C++ extension protocol called PEP - the Persistence Extension Protocol. Extension protocols [Gow97] are an Iguana/C++ concept used to provide a structured interface to the meta-level code. The PEP is defined as:

```
class PEP {
public:
    static bool init(char* filename);
    static bool close();
    static bool record(char* name, char* type, void* object);
    static bool lookup(char* name, char* type, void*& object);
    static bool remove(char* name);
};
```

PEP::record allows a symbolic name to be assigned to a persistent object. Upon successful execution, this method makes the target object a persistent root. PEP::lookup allows the recall of any previously recorded persistent root object based on its assigned symbolic name. The type argument is used to check whether the expected type and the type of the restored object match. Finally, PEP::remove allows the removal of symbolic name/object associations. PEP::init initialises the POS and a call to PEP::close results in storing all reachable persistent objects.

3.1 Using Object Persistence

An application programmer who wants to avail of object persistence simply uses the default protocol selection to select the Persistent protocol. For example:

```
defaultProtocol ==> Persistent;
```

```
class Counter {
private:
    int i;
public:
    Counter(int a) { i= a; }
    int getValue() { return i; }
    void setValue(int a) { i= a; }
};
```

will result in all instances of class Counter being potentially persistent. Within the same source file, the programmer could define other (sub)classes that would also become potentially persistent. If the programmer uses multiple source files, each source file with class declaration(s) must include the same default protocol selection statement. Note that this requirement violates type orthogonality for persistence. On the positive side, it allows the programmer not to select Persistent for classes that are transient in nature. In this case, extra care must be taken by the programmer as there is a danger that persistent object will hold references to non-persistent objects.

An example of an Iguana/C++ program that uses persistent objects is shown below. Note that the program is coded to be aware of whether it needs to create a new object or use a previously created one.

```
Counter *pc1;
PEP::init("./mypos");
if (firstTime) {
    pc1= new Counter(1);          // create object and record it in the name service
    PEP::record("/this/counter", "Counter", pc1);
} else {                          // get reference to object from name service
    PEP::lookup("/this/counter", "Counter", pc1));
}
if (pc1) {                        // use object
    cout << "Value is: " << pc1-> getValue() << endl;
}
PEP::close();
```

Note that it is not necessary to record every persistent object in the name service. If a persistent object contains references to other persistent objects, they will also be saved and will be loaded as and when required in subsequent program executions.

3.2 The Implementation of Object Persistence

In this section we describe the design and implementation of a protocol hierarchy that extends objects with persistence. We will first describe a more naive implementation that is later optimised for speed. From the base-level programmer's point of view there is no difference in the use of object persistence.

Persistent object references are different from C++ object references. They contain sufficient information to identify an object in the POS. The process of changing the references on storing/loading a persistent object is called swizzling/unswizzling. In our implementation, non-resident objects are represented by proxy objects that occupy the same amount of memory space as the objects that they represent. When a reference to a non-resident object is unswizzled, its proxy is created, if necessary, using information in the persistent reference. The persistent reference is then replaced with the appropriate address within the space occupied by the proxy. Of course, subsequent attempts to use the reference must then be caught.

The protocol hierarchy for the naive implementation is as follows:

```
protocol TypeInfo {
shared:
    reify Class      : MClass;
    reify Attribute  : MAttribute;
    reify Method     : MMethod;
};
```

```
protocol Persistent : TypeInfo {
local:
    reify Class      : PersistentClass;
shared:
    reify Creation   : PersistentCreation;
    reify Deletion   : PersistentDeletion;
    reify StateRead  : PersistentRead;
    reify StateWrite : PersistentWrite;
    reify Invocation : PersistentInvocation;
};
```

The protocol TypeInfo allows structural information to be retained about classes. This information includes the description of the class, its attribute(s) and method(s) in the form of metaobjects. The Persistent protocol overrides the behaviour of the default C++ object model for creation, deletion, state access (read/write) and method invocation. While the implementation of the Persistent protocol is complex, we will outline the role of each of the metaobject classes being used:

Class: PersistentClass extends MClass to include an object header for each persistent object. This header contains information about the object's persistent reference, state in the memory (absent/present) and the number of references that it contains (including references inherited from parent classes).

Attribute: MAttribute metaobjects are used to store information about references (e.g. type, size, access modifier, offset within the object) for swizzling/unswizzling of references in persistent objects.

Creation: on object creation, PersistentCreation checks whether the object's class has been installed in the persistent class register. If not, it installs the class and then initiates the creation of the the persistent object with an appropriately initialised meta-level and object header.

Deletion: on deleting a persistent object, PersistentDeletion checks if the object has been recorded with a name in the name service. If so, it removes the entry and then it initiates the removal of the persistent object together with its meta-level and object header.

Method Invocation: when a method is invoked on a resident object, the default method invocation is carried out. When a method is invoked on a proxy, control is passed to the PersistentClass metaobject that handles the object fault. All persistent references in the object's state are then unswizzzled.

State Read and Write: like method invocation, control is passed to the PersistentClass metaobject that is responsible for handling the object fault if the state is accessed on a proxy. Otherwise, the default state access is carried out.

3.3 Adapting the meta-level

A problem with the implementation described above is that all language operations on a potentially persistent object are trapped and carried out via the (slower) meta-level, even if the object is already present in memory and could be treated as a normal C++ object. To overcome this problem we have refined the Persistent protocol, which now takes advantage of dynamic protocol selection. The protocol hierarchy is defined as:

```
protocol Persistent : TypeInfo {
shared:
    reify Creation   : PersistentCreation;
    reify Deletion   : PersistentDeletion;
};
```

```
protocol PersistentProxy : Persistent {
shared:
    reify Class      : PersistentClass;
    reify StateRead  : PersistentRead;
    reify StateWrite : PersistentWrite;
    reify Invocation : PersistentInvocation;
};
```

The modified Persistent protocol only intercepts object creation/deletion and represents an object that is present in memory. The PersistentProxy protocol on the other hand intercedes with all other language operations and represents an object that is absent. There is a dynamic switching between the two protocols, which is initiated from the meta-level on loading a persistent object from POS.

	App. 1	*App.* 2
plain C++	1	1
Iguana Run-time check	1.21	1.51
Iguana Default	9.3	66.13
Iguana Persistent (naive)	13.0	88.7
Iguana Persistent (opt.)	1.27	1.56
Texas/C++	1.03	1.04

Table 1. Measurements showing the relative overhead of Iguana/C++ versus Texas/C++ for 2 benchmark applications.

For persistent objects that are present in memory, the Persistent is the protocol selected. Because StateRead, StateWrite and Invocation are not reified, these operations are carried out at the base-level object without involving the meta-level.

Absent objects (i.e., persistent objects that are referenced by present persistent objects but not loaded from the POS yet) are associated with the PersistentProxy protocol; all language operations are reified. When an operation is requested on a proxy base-level object, the operation is directed to the meta-level. The object fault is handled by the meta-level and the object is loaded from the POS. The POS completes the request and reference unswizzling is performed. The object is now present in memory and the meta-level configuration switches to the Persistent protocol, consequently all further operations are now carried out directly. This results in a significant performance gain.

4 Evaluation

We applied both the Persistent protocol and the Texas/C++ persistent store to two different sample applications. Application 1 creates and walks a tree of objects, application 2 calculates the number of permutations of a set of elements and is highly recursive. The measurements were conducted on a Sun Sparcstation under Solaris 2.5 [1].

The results are summarised in table 1. We carried out a number of measurements, namely

1. Run-time check: shows the costs of the run-time checks inserted into the application that allow to dynamically switch on/off reification of objects access.
2. Default: shows the costs of the Iguana/C++ default protocol, i.e. the costs of performing all of the operations via the meta-level.
3. Persistent (naive): naive implementation of persistent objects: all invocations are trapped and carried out via the meta-level.
4. Persistent (opt.): shows the overhead of the optimised implementation of the Persistent protocol.
5. Texas/C++: overhead using the Texas/C++ PStore

For all the measurements, we did not take into account the costs of object creation/deletion since they depend to a great extent on the implementation of the underlying persistent object store. The numbers show that although the overhead of reified language operations is high, we can achieve a reasonable overhead in the case where we switch off reification after the object has been loaded into memory, between 20–and 50%. This overhead consists of the run-time check and the additional costs for loading the object the first time it is accessed.

5 Concluding Remarks and Future Work

This paper described our experiences with implementing object persistence using the Iguana reflective programming language. This work was motivated by the fact that despite ongoing research in the reflection community over the last decade, meta-level programming is still restricted to a number of experimental research architectures and has not found widespread acceptance.

[1] This rather antiquated configuration was the only one that allowed the proper installation of Texas/C++.

We briefly presented the Iguana model, which offers a number of features that address some of the shortcomings found in comparable platforms, most notably the ability to augment, compose and dynamically select meta-level configurations. Our experiences show that the ability to dynamically bind objects to different meta-level configurations as provided by Iguana is an essential feature in order to gain a performance that can compare well with dedicated solutions.

With our Persistent design, and in general, full separation of concerns for object persistence is not attainable. From the base-level programmer's point of view, it seems relatively easy to avail of object persistence. The steps involved with adding support for persistent objects to existing applications using Iguana consist of associating Persistent with classes that are to be made persistent, the insertion of code to register/lookup persistent objects and an additional pre-processing phase to apply the Iguana model. It should not be denied that due to the complexity of C++ it is not trivial to provide a fully reflective language extension. So for example it is only possible to make heap allocated objects persistent due to the inability to intercede with the creation of stack allocated objects.

Meta-level programming is still a non-trivial task and requires a sound understanding of the semantics of the underlying object model. Especially debugging of meta-level code has in our experience shown to be tedious. This aside, we believe that the interface provided by Iguana allows a modular and structured development of meta-level architectures. Once useful protocols have been implemented, they can easily be used by less experienced base-level programmers.

Currently, we are working on a Remote protocol for implementing remote method invocation on reflective objects. Our plan is to make this Remote implementation interwork with Java RMI in order to directly compare performance.

Future work will investigate the composition of more complex behaviours, for example persistence, remote objects, fault tolerance and object migration. Our long term goal is to provide a library of independently composable protocols that can easily be integrated into applications using the Iguana protocol inheritance and selection mechanisms.

References

[Cah99] Vinny Cahill. Tigger: A framework supporting distributed and persistent objects. In Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors, *Implementing Application Frameworks: Object-oriented Frameworks at Work*, pages 485–519. Wiley, 1999.

[Chi93] Shigeru Chiba. Open C++ release 1.2 programmer's guide. Technical Report TR 93-3, Department of Information Science, University of Tokyo, 1993.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. In Carrie Wilpolt, editor, *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 285–299. ACM Special Interest Group on Programming Languages, ACM Press, October 1995. Also SIGPLAN Notices 30(10), October 1995.

[DSC+00] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti, and Barry Redmond. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 171–190. Springer-Verlag, Heidelberg, Germany, June 2000.

[GC96] Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. In *Proceedings of Reflection '96*, pages 137–152. XEROX Palo Alto Research Center, April 1996.

[Gow97] Brendan Gowing. *A Reflective Programming Model and Language for Dynamically Modifying Compiled Software*. PhD thesis, Department of Computer Science, University of Dublin, Trinity College, 1997.

[KFRGC98] M. O. Killijian, J. C. Fabre, J. C. Ruiz-Garcia, and S. Chiba. A metaobject protocol for fault-tolerant CORBA applications. In *Proceedings of the 17^{th} Symposium on Reliable Distributed Systems*, pages 127–134, September 1998.

[KYK+99] Mangesh Kasbekar, Shalini Yajnik, Reinhard Klemm, Yennun Huang, and Chita Das. Issues in the Design of a Reflective Library for Checkpointing C++ Objects. In *Proceedings of the 18^{th} Symposium on Reliable Distributed Systems*, 1999.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient Portable Persistent Store. In *5th International Workshop on Persistent Object Systems*, September 1992.

[SW95] Robert J. Stroud and Zhixue Wu. Using metaobject protocols to implement atomic data types. In Walter Olthoff, editor, *Proceedings of the 9^{th} European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 168–189. Springer-Verlag, August 1995.