Simplifying an Extensible Class Library Interface with OpenC++

Joël Cavarroc
ESSI -- BP 145
06903 Sophia Antipolis
France
+33 4 92 96 50 50
cavarroc@essi.fr

Sabine Moisan
INRIA Sophia Antipolis
BP 93
06902 Sophia Antipolis
France
+33 4 92 38 78 47
Sabine.Moisan@sophia.inria.fr

Jean-Paul Rigault
ESSI and CNRS/I3S
BP 145
06903 Sophia Antipolis
France
+33 4 92 96 51 25
jpr@essi.fr

ABSTRACT

In this paper we focus on the benefits that a metaobject protocol may bring to a C++ class library. This library, named FRAMELIB, implements "frames" as used in Artificial Intelligence. It addresses four important aspects: frame structure and behavior, type safety, frame instance persistence, and sophisticated debugging mechanisms.

Although frames may appear rather similar to classes as found in object-oriented languages, their implementation in C++ raises a number of technical challenges. The interface to our library introduces protocols and patterns which must be strictly followed to ensure the correctness and consistency of the code.

To simplify the usage of the library we decided to recourse to a metaobject protocol (MOP), and more specifically to OpenC++. This paper describes our experience with this MOP, the promising results that we obtained, the problems we encountered, and the design issues that this technique raises.

Keywords

C++, OpenC++, metaobject protocol, library interface.

INTRODUCTION

It is now recognized that using complex object-oriented class libraries is not as straightforward as one could expect. One reason is that a class seldom is a (re)usability unit by itself. Instead, combinations and cooperation of classes must be considered and used altogether, as it has been demonstrated with the advent of the now popular "design patterns" [7]. Another reason is that most class libraries are used through extension, that is by adding new classes, derived from or composed of the classes provided by the library. The derivation and the composition require committing to precise protocols to ensure the consistency of the new classes. The library user must add members and methods simply for the sake of making the whole class scaffolding operational.

The last remark is particularly true in a language like C++ which proposes a rich set of mechanisms to create and manipulate objects and classes. For instance, each new class derived from a library class has to comply or, at least to cope, with the corresponding mechanisms implemented in the base class. This may force the user of the library, who is the designer of the new class, to introduce specific constructors, destructor, operators, or other member-functions just to satisfy the library protocol. This is an example of the "loss of locality" that is sometimes experienced in class inheritance hierarchy.

Of course, one may leave the toil of respecting the protocol to the user of the library. However, this is not practicable if the user is a C++ novice, if the non-respect of the protocol may produce erratic behavior, if the protocol is overly complex, or if it is simply composed of too many elements. A better approach is to provide a set of tools for automatically enforcing the protocol.

We faced the problem of complex library usage protocol when we developed FRAMELIB, a C++ library of Artificial Intelligence "frames". FRAMELIB targets two kinds of users. First, human *experts*[1] define new frame types; they are seldom C++ programmers and cannot use the library as-is. Second, a set of software tools collaborates with FRAMELIB to develop knowledge-based systems and, to follow good Software Engineering practice, they should be as independent as possible from the library. For these two reasons we decided to investigate how a metaobject protocol could simplify the library interface for both experts and tools. Moreover, a metaprotocol approach helps hiding the technical programming details from the experts, who can stay at

[1] *Expert* is taken here in its Artificial Intelligence sense: the person who provides the knowledge about an application domain.

the semantic level that is relevant to them, i.e., the application one.

In this paper we first briefly describe how we have used OpenC++ to automatically handle the complex (and mostly not relevant to the expert) class library protocol. Then, we mention some problems that we encountered and we address one of the main design issues: deciding tradeoffs between the metaprotocols that C++ already provides and the ones that the metaobject protocol may supplement. In the conclusion we summarize the benefits of this experiment and we indicate some interesting extensions to our work.

OUR EXPERIENCE WITH FRAMELIB

FRAMELIB Main Characteristics

Among the various knowledge representation techniques developed for knowledge-based systems, frames [9] emerged as a widely applicable one to represent structured objects and objects interrelations. Like a class, a frame describes a collection of objects and it has attributes (known as slots). Moreover, frames can inherit from each other or be combined together to form composite objects.

The main difference with classes, though, is that frames are organized according to three levels. The *frame level* corresponds to the usual class level. The *slot level* describes the class attributes, which may refer to other frame instances. Each slot is also structured with sub-attributes or *facets*, some of which may be *daemons*, i.e., procedures called by the system without user intervention. Furthermore, to be practicable in knowledge-based systems, frames should exhibit two features not directly supported by C++: persistence (from one execution of the system to another) and type filtering (extracting instances of the same type from a collection).

The role of FRAMELIB is to provide a generic framework so that an expert may create new (domain specific) frames. Thus it is composed of high level (abstract) classes together with general mechanisms to use them and also with *patterns* [7] to build new frames. FRAMELIB meets the following requirements:

- Provide C++ classes with frame features (slots, facets, and daemons). Experts create new frames by inheritance from a common base class `Frame` (using a precise protocol), thus extending the library.
- Handle frame specific operations like persistence and type filtering.
- Ensure type safety; frames are statically typed; selection of instances based upon their type is possible.
- Use C++ features to avoid as much as possible run-time overhead, leading to an efficient implementation.
- Support sophisticated debugging mechanisms, allowing to trace back or to replay execution while staying at the expert level of abstraction.

In C++, the implementation of frames raises a number of technical challenges. Indeed if one wants to use FRAMELIB directly, the task is rather tedious and error-prone. The main reason is that C++ stands at a much lower level of abstraction than the expert's view of frames. Moreover, C++ demands numerous "housekeeping" functions[2] essentially constructors and destructor, and copy operations. Some other challenges are related to the weak power of expression of C++, especially when reflexivity is an issue. Compared to classes in C++, frames require additional "housekeeping" functions that cannot be completely specified at compile-time: inspection (for debugging and persistence), instance type filtering, method-call trapping (for debugging), etc.

The OpenC++ Metaobject Protocol in FRAMELIB

To automatically enforce the library protocol there are several possible approaches: let a translator do it entirely, use macro-processing facilities to do it, modify the C++ compiler to help doing it, extend the C++ language to handle frames, or let a metaobject protocol (MOP) [2] do it automatically. We chose to use the metaobject protocol of OpenC++ version 2 [3] to automatically generate the "housekeeping" functions.

In MOP models, a *program* defines language extensions. MOPs make it possible to introduce new syntactic forms or new semantic behaviors. They also allow modifying built-in mechanisms, like method calls or instance creations. Thus a MOP combines all previously mentioned approaches: it provides a kind of clever macro-processing facility, it extends the language, and it produces a compiler for the extended language.

In a compiled language like C++, there is a clear separation between compile-time (where classes and types are known) and run-time where most of the information about types has been lost. Depending on the MOP, the metaobjects exist at compile time, at run-time, or both. For example, Chiba's OpenC++ version 1 [2] was a run-time MOP whereas OpenC++ version 2 [3] is a compile-time MOP. Yet, a compile-time MOP allows creating metaobjects that will be present at run-time.

In FRAMELIB the MOP automatically generates the frame constructors and destructor, as well as most housekeeping functions. It is also responsible for extending C++ to support the following (frame) features: frame syntax and semantics, especially the definition of slots; persistence handling; class inspection for trace and debugging. In addition, through run-time metaobjects, it prepares frames for persistence, debugging, and type filtering.

---

[2] By "housekeeping", we mean either functions which are indispensable for the internal mechanics of the classes but which lay at a much lower level of abstraction than the application itself, or functions which are pure programming artifacts, like printing.

The MOP defines an extended language close to the experts' frame description. Despite its C++-like syntax, it is rid of the "housekeeping" peculiarities of C++.

There is only one metaclass in our application, named `FrameClass`. Several metaclasses could have been useful and more elegant, but we had to cope with some current OpenC++ limitations that we shall explain later (see section "Limitations of OpenC++"). So class `Frame`, the base class of all frames in FRAMELIB, is declared to have `FrameClass` as its metaclass and consequently, all classes derived from `Frame` will inherit the same metaclass.

Features Supported by `FrameClass`

Syntactic Extensions

The OpenC++ MOP allows to slightly modify the syntax of C++: basically it is possible to add new type and access modifiers, and new keywords. We use this facility to handle slot and facet definitions. For instance, the following example presents the definition of a frame class representing a car using the extended language (new keywords appear in boldface). OpenC++ will eventually translate this code into plain C++.

```
class Car : public Frame
{
public:
  int get_from_licence() const;
slot:
  range("Renault", "Ferrari",
    "Honda", "Mercedes") String brand;
  default_value(150) range(100-300)
    int max_speed;
  if_needed(get_from_licence) int age;
  Engine the_engine;
};
```

Class (frame) `Car` has four slots brand named `max_speed`, `age`, and `the_engine` of respective types `String`, `int`, `int`, and `Engine` (likely an other expert-defined frame). The example shows several types of facets (see section "FRAMELIB Main Characteristics"): two ranges (an enumeration and an integer interval), a default value, and a daemon[3] associated to the local public method `get_from_licence` . These slot facets are denoted by C++-like type modifiers.

Frame Semantics and Inspection

Metaclass `FrameClass` generates the C++ constructors needed to make the slots operational. But its most important role is to provide new frame classes with frame semantics, and especially with inspection and persistence. These features are implemented by overriding the OpenC++ metamethod `TranslateSelf`.

For inspection, the MOP connects each frame instance with its type descriptor and creates two new methods for each frame class: `display()` that lists the (frame) class slots with their types and `inspect()` that does the same thing but adds the current value of the slots. `FrameClass` also adds all methods needed to dynamically handle frame type information: `isA(), isKindOf(), isSuperOf()`. These methods make it possible to filter collections of frame instances through a type criterion.

Handling persistence is slightly more complicated. Saving frames to disk is the simpler part: the MOP just adds a new method, `save()`, which performs a memberwise (and slotwise) saving. The definition of the slot template classes includes a `save()` member-function. The previous protocol is applied recursively to the components of the class. It has been inspired by the memberwise default "copy constructor" (or assignment operator) built in C++. Note that the `save()` method does not only save value information; it also records type information. Restoring is the trickier part, since the type of the object being restored is known at run-time only. So we need a kind of "virtual construction" which is not directly supported by C++. FRAMELIB uses the "exemplar idiom", as described by James Coplien [5] to implement a virtual construction protocol. This virtual construction mechanism is almost totally implemented in pure C++; the MOP has only to introduce the definitions of methods performing instance restoration.

DESIGN ISSUES

Limitations of OpenC++

We started this work with versions 2.3 and 2.4 of OpenC++, which suffered a number of unimplemented syntactic features. The latest version (2.5) of OpenC++ [3] is now rid of these limitations.

More problematic was the design of FRAMELIB metaclasses. In a first attempt, we defined three metaclasses, each of which was responsible for a particular aspect of the frame features: one for inspection, the second for persistence, and the last for frame semantics. Each metaclass added new features to their classes, thus modifying the classes syntax trees. But the resulting augmented classes have to be analyzed by the remaining metaclasses. Unfortunately, in OpenC++ v2, when a metaprogram modifies a class definition, the resulting code is pure C++, not OpenC++[4]. This means that it cannot be analyzed again by the metaprogram. The only possibility with OpenC++ as it is by now is to stack metaprograms, that is to pipe several independent compilers. This clearly lengthens compilation time. For now, we decided to simplify our design and to delegate all MOP-related features to a single metaclass, `FrameClass`.

---

[3] This so-called "if needed" daemon is invoked each time a slot with no current value is accessed and provides a computed value.

[4] This limitation of the OpenC++ MOP might be fixed in future releases.

Tradeoffs between OpenC++ and C++ own Metamechanisms

There is a more fundamental and conceptual design problem. C++ already proposes a number of built-in "metafeatures": static members and methods, construction and destruction protocols, default memberwise copy operations (copy constructor and assignment operator), Run Time Type Information (RTTI), and, last but not least, templates. These built-in protocols make it possible for programmers to implement new "metafeatures". The "virtual construction" mechanism mentioned in the previous section is an example of those.

This immediately raises several issues. How to distribute the roles between the built-in protocols and the MOP-defined ones? What features should be implemented directly at the C++ level? Which ones should be entirely delegated to the MOP? Which tradeoffs between these different metalevels?

We have no general solution to this problem. Indeed, in the design of FRAMELIB, we gave different answers to different situations, yet complying with the following policy. Each time a C++ programmer would have to define a feature (class, method, instance) to respect the library protocol (that is the internal "housekeeping"), we tried to have the feature defined automatically instead, via the MOP. By contrast, if C++ could do it *directly* (although sometimes partially), let C++ do it. *Directly* means here: without loosing C++ virtues, especially without fighting against the type system, without run-time overhead, without maintenance overhead (this means that the corresponding feature could be wired up, once for all, in the library, and that the user could be oblivious of it).

A significant example is slot access support and daemon triggering (see section "FRAMELIB Main Characteristics"). This could be done with OpenC++ MOP, since the pair of metamethods `TranslateMemberRead` and `TranslateMemberWrite` trap all accesses to class data members. This can also be done purely within C++ by overriding the assignment and the function-call operators. The latter solution being a *direct* implementation in the previous sense, we chose it.

In the persistence support, we use both direct C++ code (the exemplar idioms) and the MOP to generate the exemplar definitions and the restoring methods of expert-defined frame classes. This appeared more convenient and more efficient, since most of the task can be done at the C++ level, the MOP being in charge of the tedious, error-prone, "not-to-forget" part.

As a last example, since no inspection mechanism is available in C++, the support of inspection has to be entirely imparted to the MOP, without any question.

We have no doubt that the tradeoffs between the language level and the MOP level are worth studying, and that this will raise some new issues in the design of the object-oriented languages themselves.

CONCLUSION

So far, MOPs over languages such as C++ have been mostly used in the domain of distributed applications, concurrency, or databases [1,4,6,8]. Our purpose was different: to enforce independence between a complex C++ library and the tools that use it, as well as to facilitate the expression of experts' knowledge.

Despite the relative young age of MOPs for C++, our experience indicates that the approach is feasible and promising. The independence between library and tools, the possibility of concurrent evolution of both, the ease of maintenance are real benefits of the MOP. Besides, due to the characteristics of OpenC++, these benefits do not incur any performance penalty.

The use of MOP that we presented here is simple. However, we see at least three main advantages to our approach. First, the MOP enforces modularity among rapidly evolving tools. Second, owing to the MOP, FRAMELIB is self-contained and safe to use directly. It is type-safe, as any C++ library should be. The code generated through the MOP is guaranteed to be complete; there is no risk of forgetting an important feature that would break up the protocol for using the library. Furthermore, the library is open to extensions, at the simple cost of modifying or extending the metaprogram. Lastly, the MOP is only in charge of the rough C++ technical particulars, whereas the rest of the code remains at the expert level. Because of this separation of concerns, the abstraction levels are well identified and the code is clear and rid of unnecessary low-level details.

After this feasibility study, we are convinced that MOPs can be used for purposes beyond simple code generation. We plan to use OpenC++ more systematically: for instance to reify frame relationships and to tune knowledge-based system inference engines.

As Software Engineering more and more focuses on reuse and component assembly, the libraries, toolkits, or frameworks tend to become more complicated. A challenge is then to simplify the protocols to use them. The internal details of the library should be hidden from the user and taken care automatically. The interface with the library should be *programmable*. We strongly believe that metaobject protocols techniques initiate new tracks to improve extensibility, versatility and openness of component libraries.

ACKNOWLEDGMENTS

The authors are grateful to Shigeru Chiba for his availability and his eagerness to improve the OpenC++ system.

REFERENCES

1. Caromel, D., Belloncle, F., and Roudier, Y. The C++// System, in *Parallel Programming Using C++.*, MIT Press, 1996.

2. Chiba, S. A Metaobject Protocol for C++, in *Proceedings. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (1995), vol. 30 of SIGPLAN Notice, ACM, 285--299.
3. Chiba, S. Open C++ 2.5 Reference Manual. Institute of Information Science, University of Tsukuba, 1997.
4. Chiba, S., and Masuda, T. Designing an Extensible Distributed Language with a Meta-Level Architecture, in *Proceedings of the 7th European Conference on Object-Oriented Programming* (ECOOP'93) (Kaiserslautern, July 1993), 482--501.
5. Coplien, J. O. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, 1992.
6. Fabre, J. C., and Perénnou, T. FRIENDS -- A Flexible Architecture for Implementing Fault-Tolerant and Secure Distributed Applications, in *Proceedings of the European Dependable Computing Conference* (Taormina, Italy, Octobre 1996).
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publ. Co, 1995.
8. Gowing, B., and Cahill, V. Meta-Object Protocols for C++: the Iguana Approach. In *Proceedings of Reflection 96* (1996), 137--152.
9. Minsky, M. *A Framework for Representing Knowledge*. Readings in knowledge representation. (1985).