Date of acceptance Grade

Instructor

Introduction to the basic concepts of computational reflection

Alexander Tamm

Helsinki 9th October 2003

Seminar presentation

UNIVERSITY OF HELSINKI
Department of Computer Science

Abstract

Computational reflection (hereafter *"reflection"*) is the ability of a programme to observe or change its code and aspects of its programming language at runtime. This paper is an introduction to the basic concepts of reflection. Different methods and aspects of reflection are discussed. *Reification* is defined as a method by which a programme can express aspects of itself or the programming language used. Aspects of computer programmes can be observed and acted upon using *inspection*.

Reflection in computer programmes are supported by languages with *reflective architectures*. Two distinct styles of reflective architectures can be recognised, *procedural reflection* and *declarative reflection*. Declarative reflection is the ability of a language to reify a running program and abstract data types used. Procedural reflection is the method by which a programme can affect aspects of the programming language used, i.e.. change the meaning of the operation. A new language *L'* is formed, as it were, based on - and expressed in - the original language *L* used. Since aspects of *L'* can be modified and a new language *L"* constructed, we stumble upon *reflective towers*, which are also discussed in this paper.

Object-oriented programming can receive a new aspect of abstraction through reflection. Object describing objects, or *meta-objects*, can be used. Meta-objects are discussed, as well as *meta-object protocols* or *MOP*.

Contents

1 Introduction

Within the programming language community, one area of study is that of *computational reflection*. It has been noted that it could be beneficial for a computer programme to be provided with information about itself and its running environment. Such information allows the programme to adapt to its environment and react to outside circumstances. Computational reflection also provides the programme with a way of expressing itself to outside components. This is the concept of *reification*.

Traditionally, the general opinion has supported the view that software (programmes, objects, libraries) should implement a "black box", i.e.. the software receives input and gives output. What goes on inside it is hidden from both users and other software. Blair & Coulson express the opinion that this approach is not always possible or even desirable [BlC]. They argue that software can possess vital information for the efficient or practical execution of some software. Indeed, they propose a *principled approach* to open implementations. (A language with reflective facilities is said to be open-ended [Mae87], i.e.. it supports open implementations.)

One very interesting use of computational reflection is the ability of the programme to alter the functionality of the language it is written in. This ability is called *procedural reflection* and is one of two main types of *reflective architectures* presented in this paper. Reflective architectures are, basically, the architecture of a programming language that support computational reflection. Procedural reflection has some drawbacks, mainly performance and *reflective towers*. These are stacks of interpreters that are necessitated by the fact that a modified language needs an interpreter running on a lower-level interpreter of the unmodified language. We shall see that such stacks are potentially dangerous for the performance of the programme, but can be very useful when used with care. As a contrast to procedural reflection stands another type of reflective architectures. In these architectures, the self-representation of a system is different from its implementation. The difference between these two types of architectures is not a clear one, and we shall see that it is not always practical to make the distinction.

Computational reflection is a very natural feature for the field of object-oriented programming. Today, a programme can be running on several different platforms, many of which are specifically designed to be independent of the underlying hardware. The Java Virtual Machine is a well-known example, but it is generally regarded to be tied to the Java programming language. Another example is Microsoft's .NET, which has its Common Language Runtime, which is not only hardware-independent, but also language-independent. Such a platform can only support the least common denominator of the programming languages it supports and features of some languages might not be available on the platform.

2 Computational reflection

Maes defines computational reflection *to be the behaviour exhibited by a reflective system, where a reflective system is a computational system which is about itself in a causally connected way.* [Mae87] Further, Maes describes a system to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that changes in one of them are reflected upon the other one. For example, suppose the position of a robot-arm in a factory is changed because of some environmental factors. The system steering this robot-arm probably has some data-structures representing the position of the arm. The new position of the arm must be visible in the data-structures in the steering system. In the same way, when the data-structures are modified by the steering-system, the changes cause the robot-arm to change its position. The steering-system and the robot-arm are then linked in a *causally connected way.*

Reflective computation is in many ways a part of every-day computational systems, according to Maes and support for reflection should be integrated as a fundamental tool in programming languages [Mae87]. One example is the frequent use of printing functions for debugging. Instead of riddling the code with e.g. `printf()` statements, the software engineer could rely on features in the programming language itself to keep him/her informed about what the programme is doing.

2.1 Reification

One important aspect of computational reflection is the ability of a programme to express aspects of itself, or present meta-information. This ability and the process is called *reification*. Reification is defined by Malenfant et al. in [MJD96] as the process by which a program can express aspects of itself or the language in which it is written. The aspects are expressed in the actual language used. It is stressed that the reification data are causally connected to the reified information. In other words, changes in one should be fully represented in the other.

Reification in an object-oriented language is seen by Madany et al. as *the representation of an attribute of an object-oriented programme such as member function dispatch, inheritance and object structure as an object within that programme.* Further: *Such a reifying object provides member functions that support reflective computation or reflective update* [Mad92]. This definition is, however, influenced by the perspective of the authors, which is an operating system perspective [Mad92].

2.2 Inspection

An interesting use for reflection, from a software engineers perspective, is the ability to inspect the execution of code and the state of objects. The practical uses are mostly for debugging software, but also for testing it. Anybody who has written software of any significant size knows the value of debugging and testing software.

Traditionally, inspection takes on the form of heavy use of functions for printing information. Some of the drawbacks to this are, for example, filling up the code with no significance for the end-product and having separate code for debugging/testing and for production use.

2.3 Reflection

Malenfant et al. promote the idea that true reflection imposes no limit on what the programme may observe or modify [MJD96]. This is stated as a long term goal. However, the authors admit that this is, not only practically, but also theoretically limited, due to e.g.. paradoxical situations. Basically this sounds like the Swiss army knife of programming, but not everybody shares this view. For example, Madany et al. recommend against the reification of instruction code [Mad92]. Maes states that reflection does not contribute to the solving of a problem directly, but rather contributes to the internal organisation of a system or to its interface to the surrounding worlds [Mae87]. The exact definition of what is reflection and what isn't, is further blurred by the fact that some systems already show some aspects of reflection, but not necessarily all. Maes further states that programming languages today (1987, NB) do not fully recognise the importance of reflective computation.

3 Reflective architectures

According to Maes, a programming language is said to have a reflective architecture if it recognises reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly [Mae87]. In such a language, the interpreter has to give a running program access to data representing the programme itself. Programmes then have the possibility to perform reflective computation on themselves. Furthermore, the causal connection between the data and the aspects of the programme must be guaranteed. In other words, modifications made by the programme most be reflected in its status and its computation.

Maes identifies two distinct styles of reflective architectures [Mae87], *procedural reflection* and *declarative reflection*. The difference between these architectures is in the self-representation. In procedural reflection, the self-representation is actually an implementation of the system, whereas in declarative reflection, the self-representation is more a collection of meta-data. However, the distinction between these two architectures should rather be viewed as a continuum, according to Maes [Mae87]. In other words, a system can support either one, or both, or possibly some form that has characteristics from both.

3.1 Procedural reflection

A reflective architecture that allows the modification of the language itself, is said to support procedural reflection. This is most naturally achieved in an interpreted language. An interpreter of such a language, is theoretically able to change its own semantics. Such an interpreter is called a *m*eta-circular interpreter.

The causal link between the system and its self-representation is guaranteed, since the self-representation is the implementation. This is a feature of the meta-circular interpreter. It is worth noticing that a meta-circular interpreter requires that programmes can be viewed as data-structures of the language. Such a requirement is basically trivial in most programming languages that are text-based. Such a language need only support a character string data-type and a function for evaluation code at run-time, e.g.. `eval()` .

One of the drawbacks of procedural reflection is lack of efficiency. The reflective computations need the self-representation itself as data in such a way that reasoning about the system is practical. At the same time, the self-representation is the implementation and therefore impacts in efficiency are often unavoidable.

3.2 Declarative reflection

The problems with procedural reflection has, according to Maes, led to the development of a different reflective architecture [Mae87]. Maes defines these architectures as supporting *declarative reflection*. This type of reflective architecture is different from procedural reflection in that it separates the implementation of the system from the self-representation of the system. In other words, such an architecture is not concerned with allowing programmes to alter its running environment, but rather present information about itself. The programme could, for example, impose restrictions and requirements that the system has to fulfil.

It is noted that declarative reflection can generally be regarded as easier to implement efficiently [MJD96]. Malenfant et al., who prefer to call it structural reflection, also note that declarative reflection is more suitable for compiled languages than is procedural reflection (a.k.a. behavioural reflection). The main questions about procedural reflection in compiled languages are [MJD96]:

1. How to represent the language semantics in such a way that programmes can modify it?
2. How to represent the run-time data in such a way as to provide programmes with easy access to them while retaining efficiency?
3. How to compile programmes where the semantics of the language can be modified?
4. How to compile programmes where the implementation of the run-time system can be modified?

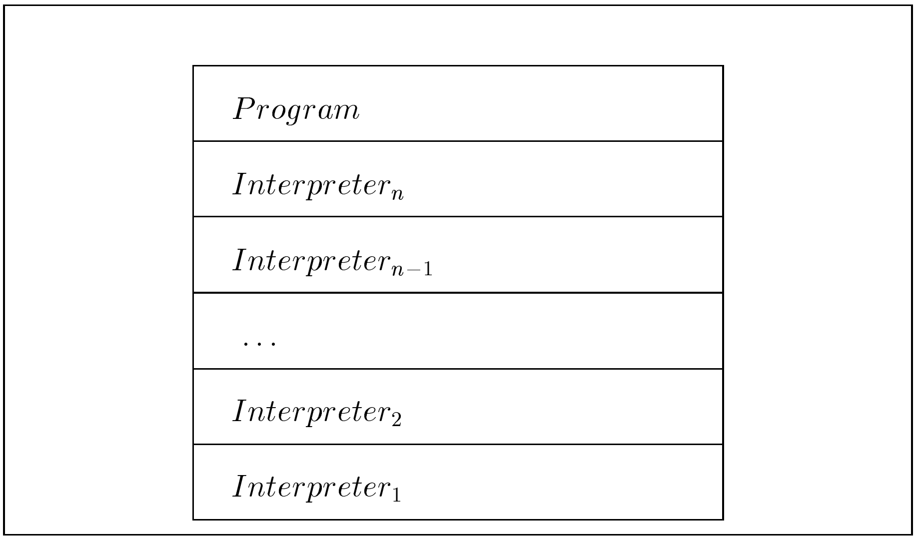

Figure 1: A reflective tower

3.3 Tower of reflection

In a language that supports procedural reflection, a programme P can alter the semantics of it's programming language L and the result is a new language L', which is both based on and expressed in L. Support for L' must be provided by either a compiler or an interpreter. If L is an interpreted language, the code expressed in L' can for example be interpreted by language constructs such as `eval()`. In any case, P must be running on an interpreter I' for the language L'. The interpreter itself is already running on I, which interprets L. Changes to L' result in yet another language L'' and hence a new interpreter I'' is required. The stack of interpreters running interpreters is theoretically infinite. This stack is called a *reflective tower* or *tower of reflection*. See figure 1 for an illustration of a reflective tower.

Malenfant et al. recognise that although the height of the reflective tower, or it's *degree of introspection* is theoretically infinite, a well-defined reflective program never uses an infinite level of reflection [MJD96]. Reflective towers are compared to recursive functions. As any first year student of computer science knows, recursive functions calls are potentially infinite, but a powerful programming idiom, when used with care. In much the same way as a well-defined recursive function call will end in an orderly fashion, the well-defined reflective program will ultimately meet a non-reflective interpreter.

4 Reflection and object-orientation

According to Maes, computational reflection fits most naturally in the domain of object-oriented programming [Mae87]. In the field of object-oriented programming, data-abstraction has always been an issue of highest interest. Since an objects innards are hidden from its surroundings, it is natural to let the object itself reflect upon its functionality. There are several issues about the running environment of an object in an object-oriented language where systems are different. This is due to the fact that the OOL community has still not agreed upon what constitutes object-orientedness. There was no agreement in 1987 [Mae87] and there still isn't. For example, an object might need to know whether or not multiple inheritance is allowed. C++ allows multiple inheritance, but the CLX-objects of Borlands compiler suites do not, since they are originally developed in Delphi. Microsoft's .NET platform also has support for C++, but does not allow multiple inheritance. Other issues are how objects are created and how initialisation is handled.

There are some examples of object-oriented languages with support for reflective computation, but these are generally not in wide use. This is mostly due to the fact that reflection has never been the primary concern of programming language designers. Reflection has often been given the back seat and other features have been favoured, for example object orientedness, simplicity or adaptability to a certain task.

One rather interesting example of a programming language that supports both object-orientedness and reflection is Perl. In itself, Perl is not an object-oriented language and provides little or no support for object-orientedness in its definition. However, classes, objects and inheritance are nowadays a widely used feature, through the `bless()` function [Perl]. Being an interpreted language, perl provides the software engineer with the ability to define classes on the fly and, even, redefining classes to work in a new way. The actual instruction code of member functions can thus be available to the object itself.

Support for reflection is provided by *meta-objects* in some object-oriented languages. Examples are the *CLOS MOP* and *Open C++* [ChMa]. (*CLOS = Common Lisp Object System, an object-oriented programming language embedded in and somewhat integrated into Common Lisp* [CLiki].)

Madany et al. point out that C++ supports some forms of reflection, since it allows the overloading of the operator `new()` [Mad92]. This can be used to implement a reference count, which is necessary for the successful implementation of a garbage collector. This has also been used by Stroustrup as an answer to the criticism that C++ has received due to the lack of a garbage collector integrated in the language [Str85-97]. It is interesting to note that while Madany et al. do not endorse trying to modify C++ to have a fully reflective architecture, they suggest that C++ might benefit from some reflective mechanisms [Mad92]. These mechanisms include the run-time presentation of an objects class, the structures of instances of a class and the superclass's and subclasses of a class.

4.1 Meta-objects

Information about an object can conveniently be stored in another object. A pointer to this *meta-object* is stored in the object itself. The meta-object can provide means for reflection whenever the object is used in any way. For example, whenever a method is called in an object, its corresponding "meta-method" can be called in the meta-object. The actual semantics of the meta-method can be up to the implementor. One use would be simple reification of the method-call, such as printing the information that the method is being called and what arguments it is being called with. Another use could be to monitor some status of the object or keep a reference count.

To the responsibilities of the meta-objects include more than simply reacting to method calls. Other uses are, for example, the reification of the names of member functions and member variables, the types and functionalities of the members, and so forth.

Several questions arise when discussing meta-objects. Should all meta-objects have a corresponding meta-class, as objects have classes? Can these meta-classes be generated by the translator of the (reflective) programming language? Should the meta-classes have the same super-class - sub-class relationship as their corresponding classes? The existence of meta-meta-classes have also been put forth. This idea is, however, impractical. When does the level of meta-ness end?

4.2 Meta-object protocols

For a programming language to support computational reflection by meta-objects, some sort of meta-object protocol, or MOP, is needed. As shown by Chiba & Masuda, it is not necessary to include reflection in the language itself, it can also be implemented independently [ChMa]. The idea of the meta-object protocol is to provide for simple-to-use facilities that support reflection. Instead of having to write calls to methods in meta-objects, the programmer should just have to express "This object or member function should be reified."

Chiba & Masuda defines a meta-object protocol as *a meta-protocol organised using object-oriented techniques* in [ChMa] (footnote). They also present a proposal for a meta-object protocol, for the C++ programming language. The MOP is called *Open C++* and extends the functionality of "vanilla" C++. MOP-instructions in *Open C++* are preceded with the line-comment (double slash, //) of C++ and "MOP". In this way, the actual C++ code is still valid C++ code and the MOP-instructions are interpreted by an independent MOP-compiler.

Madany et al. describe the *Choices* project in which reflection is provided to C++ [Mad92, Cam89]. Reflection in *Choices* supports:

Storage Concurrency and performance of memory allocation and deallocation.

Existence Mechanisms for automating the deletion of objects.

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 representation of objects for tracing, debugging and instrumentation.

The reflection provided by *Choices* clearly shows that it is designed to serve a programming language (C++) out of an operating system perspective. The needs put forth for *Choices* is not the same as the needs put forth for other systems. For example, Madany et al. argue that encapsulation of data is worthless, unless hardware-enforced, a feature of *Choices*. The difference of opinion with Blair & Coulson about openness is notable.

5 Conclusive remarks

This paper has introduced the basic concepts of computational reflection. We have presented such terms as reflection, reification and inspection. Different reflective architectures have been discussed, mainly procedural reflection and declarative reflection.

Computational reflection in the domain of object-oriented programming has been discussed and the use of meta-objects described. Meta-objects have been linked to the use of meta-object protocols and two MOP's have been briefly described and a third mentioned.

Benefits of computational reflection have been explained, among them the increase in adaptability and better possibilities for debugging and testing software. Problems of reflection have also been presented, such as the impact on performance and the occurrence of reflective towers.

Maes complained in 1987 about the lack of reflective architectures in programming languages and put forth some good points for the case of computational reflection [Mae87]. Today, sixteen years later, computational reflection is still not as profoundly present in programming languages as Maes wished it to be. Several languages incorporate some features of reflection, but none incorporate them all. At least not any of the languages in wide practical use. Is this a sign that perhaps computational reflection isn't quite the holy grail as Maes put it? Or is it simply the case that the real need for reflection hasn't surfaced yet? Perhaps the answer lies somewhere in between. Today, the Internet has connected more and more computers, and hence users, together and the need for distributed applications and the integration of several different services is ever-increasing. Perhaps the programming language community has been lagging after and not implementing what it should already be prepared for.

References

BlC Gordon S. Blair & Geoff Coulson, The Case for Reflective Middleware, Department of Computing, Lancaster University, Bailrigg, Lancaster, U.K.

ChMa Shigery Chiba & Takashi Masuda, Designing an Extensible Distributed Language with a Meta-Level Architecture, Department of Information Science, University of Tokyo.

CLiki http://cliki.tunes.org/CLOS

Mad92 Peter W. Madany, Nayeem Islam, Panos Kougioris & Roy H. Campbell, Reification and reflection in C++: An operating systems perspective, University of Illinois at Urbana-Champaign, Department of computer science, Urbana, Illinois, USA, 1992.

Mae87 Pattie Maes, Concepts and experiments in computational reflection, AI-LAB, Vrije Universiteit Brussel, Brussels, Belgium, 1987

MJD96 J. Malenfant, M. Jacques & F-N. Demers, A Tutorial on Behavioral Reflection and its Implementation, Département d'informatique et recherche opérationnelle, Université de Montréal, Montréal, Québec, Canada, 1996.

Perl Larry Wall et al., man perlboot, man perltoot, man perlobj

Cam89 Roy H. Campbell, Gary M. Johnston, Peter W. Madany & Vincent F. Russo, Principles of object-oriented operating system design University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1989.

Str85-97 Bjarne Stroustrup, The C++ programming language 1st-3d ed., Addison-Wesley