Computational Reflection

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Outline

1 Computational Reflection
   - Background.
   - General Definitions.
   - Reflective Systems: Definition, Characterization, Classification.

2 Reflection and Object-Oriented Programming Languages.
   - Structural and Behavioral Reflection.
   - Reflective Models: Meta-Class, -Object and Communication Reification.

3 Bibliography

Background.

General Definitions.

Reflective Systems: Definition, Characterization, Classification.

Reection.

Historical Overview.

In the sixties
   - Research field: artificial intelligence;
   - first approaches to reflection: intelligent behavior.

In the eighties
   - Research field: programming languages;
     - Brian C. Smith, he introduces the reflection in Lisp (1982 and 1984), the reflective tower has been defined;
     - several reflective lisp-oriented languages have been defined (they exploit the quoting mechanism).

In the meanwhile
   - Research field: logic programming;
     - the meta-programming takes place in PROLOG.

In the eighties and the nineties
   - Research field: object-oriented programming languages;
   - Pattie Maes defines the computational reflection in OOPL (1987);
   - SmallTalk is elected as the best reflective programming language.

In the nineties
   - Research field: typed and/or compiled object-oriented programming languages;
   - Shigeru Chiba realizes OpenC++ (1993-1995), and OpenJava (1999):

In the 1997
   - Gregor Kiczales et al. define the aspect-oriented programming and the story ends.
Computational Reflection. 
General Definitions.

Computational reflection can be intuitively defined as:

"the activity done by a SW system to represent and manipulate its own structure and behavior". [1]

The reflective activity is done analogously to the usual system activity.


Computational Reflection. 
Reflective Systems.

From the definition, we could evince that a reflective system is:

- a software system logically layered into two or more levels respectively called base-level and meta-levels;
- the system running in a meta-level observes and manipulates the system running in the underlying level (reflective tower).

Characteristics

- the system running in the base-level is unaware of the existence and of the work of the systems running in the overlying levels;
- a meta-level system acts on a representation (called reification) of the system running in the underlying levels; and
- a system and its reification are causally connected and therefore, they are kept mutually consistent.

Pattie Maes has pioneered the field:

- a computational system is a system that can reason about and act on its applicative domain;
- a computational system is causally connected to its domain if and only if a change to its domain is reflected on it and vice versa;
- a meta-system is a computational system whose applicative domain is another computational system;
- reflection is the property of reasoning about and acting on itself; therefore
- a reflective system is a meta-system causally connected to itself.

The reflective systems can be classified on:

The kind of reflective actions that the system can carry out:
- structural and behavioral reflection;
- introspection (just to observe) and intercession (to alter).

When the meta-level entities exist:
- compile-time;
- load-time; and
- run-time.

The abstraction level:
- in Java, the meta-level entities can represent either the source or the bytecode at load-time.

The behavioral reflection allows the program of monitoring and manipulating its own computation, e.g.:
- to trap a method call and activating a different method instead;
- to monitor the object state;
- to create new objects, and so on.

All these reflective activities can take place at run-time without a specific support.

The structural reflection allows the program of inspecting and altering its own structure, e.g.:
- the code of a method can be modified or removed from the class.
- new methods and field can be added to a class, and so on;

All these reflective activities need a specific support by the execution environment (either from the virtual machine or the run-time environment) to be carried out at run-time.

The base-level entities (referents) are reified into the meta-level, i.e., they have a representative into the meta-level.

Such a representative, called reification, has to:
- support all the operations and have the same characteristics of the corresponding referent;
- be kept consistent to its referent (causal connection);
- be subjected to the manipulations of the meta-level entities to protect the base-level entities from potential inconsistency.

Any change carried out on the reification has to be reflected on the corresponding referent.

Jacques Ferber [2] has raised some issues that the developers must take in consideration:
- which kind of entities should be reified?
- what and how it is implemented the causal connection?
- when does the execution shift to the meta-level?

Computational Reflection
Which Kind of Entities Should Be Reified?

It depends on the programming language:
- **functional**: lambda expression/closures, environment, continuations, and so on ...
- **object-oriented**: objects, methods, classes, messages and so on ...
- **concurrent and object-oriented**: threads, processes, schedulers, monitors, and so on ...
- **distribution**: namespaces, proxies, mailers, and so on ...

Computational Reflection
What and How It Is Implemented the Causal Connection?

It depends on when the reflective activities take place:
- **at run-time**: the causal connection is explicit and must be maintained by an entities super-parties, e.g., by the virtual machine or by the run-time environment;
- **at compile-time**: the causal connection is implicit, base-level and meta-levels are merged together during a preprocessing phase;
- **at load-time**: in this case the causal connection behaves as in the case, reflection takes place at compile-time.

Most of the times, the supported reflective activity is related to observe (introspection) the base-level system so the causal connection become unilateral and can be managed by the meta-entities.

Computational Reflection
When Does the Execution Shift to the Meta-Level?

Switching among levels depends on:
- which entities are reified;
- when such entities are reified; and
- how the causal connection is managed.

The switch is logically realized by two actions: the **shift-up** and **shift-down** actions.

When
- an observed element changes; or
- an action is going to be done;

the computational flow passes into the meta-level (**shift-up**).
Instead
- the computational flow goes back (**shift-down**) on the meta-level program decision.

Usually, the **shift-up** action is managed by call-backs.

Walter Cazzola
Computational Reflection Slide 13 of 28

Computational Reflection
Reflection and Object-Oriented Programming Languages.

Structural and Behavioral Reflection.
Reflection Models: Meta-Class, Meta-Object and Communication Reification.

Every couple <data, program> is encapsulated into an object.

It is possible to reify the program components and its state as first class objects.

A basic characteristic is:
- self, this, ..., therefore an object has a certain degree of awareness of itself.

Walter Cazzola
Computational Reflection Slide 16 of 28
Structural Reflection
- objects creation and initialization
  - constructors;
  - prototype;
  - meta-classes;
- class manipulation
  - to add or remove fields and methods;
  - to change the super class;

Behavioral Reflection
- message sending or application
  - classes and inheritance;
  - prototypes and delegation;
  - errors;
  - encapsulations;
  - proxies;
  - meta-objects;

The objects running in the meta-level, called **meta-objects** are associated to all (or just to someone of) the objects running in the base-level, called **referents**.

The connection among referents and meta-objects is called causal connection when it is a two-way link or meta-connection when it is a one-way link.

The meta-objects exist at run-time and extend or modify the semantics of some mechanisms:
- method invocation, field access, object creation, and so on.

The **MOP** is the set of messages that a meta-object can understand.

Each method call is logged into a file.

The objects running in the meta-level, called **meta-objects** are associated to all (or just to someone of) the objects running in the base-level, called **referents**.

The connection among referents and meta-objects is called causal connection when it is a two-way link or meta-connection when it is a one-way link.

The meta-objects exist at run-time and extend or modify the semantics of some mechanisms:
- method invocation, field access, object creation, and so on.

The **MOP** is the set of messages that a meta-object can understand.
The meta-class based approach.
- the classes carry out the reflective activity;
- the reflective tower is realized by the inheritance link.

Drawbacks.
- all the instances of a class share the same meta-class therefore
  the same reflective behavior (the granularity of reflection is at
  the class level);
- the classes have to be available at run-time.

Programming Languages.
- SmallTalk (Adele Goldberg, 1972);
- ObjVLisp (Pierre Cointe, 1987);

The meta-object approach.
- some special objects instantiated by a special class are asso-
  ciated to the base-level objects, they deal with the reflective
  computation;
- the reflective tower is realized by clientship.

Drawbacks and Benefits
- the granularity of reflection is at the object level;
- it cannot manage object communications, the approach lacks of a
  global view of the communication.

Programming Languages
- CCEL (Carolyn Duby, 1992), Iguana (Brendan Gowing and Vinny
  Cahill, 1996);
- ABCL-R (Akinori Yonezawa and Satoshi Matsuoka, Actors meet
  Reflection, 1988);
- OpenC++ (Shigeru Chiba <2.0, 1993)
**Approach to the reification of the communication.**
- some special objects reify the messages exchanged among the base-level objects, these special objects deal with the reflective computation.

**Drawbacks and Benefits**
- the granularity of reflection is at the level of method call (very flexible);
- it is possible to reflect on the whole message exchange (global view);
- there is a meta-entities proliferation; and
- the lifecycle of the meta-entities is strictly tied to the lifecycle of the message exchange (lost the history of the reflective computation).

**Programming Languages**
- Mering (Jacques Ferber, 1987);
- CodA (Jeff McAffer, 1994), mChaRM (Walter Cazzola, 1997).

---

**Conclusions.**
- it permits to open up a system to postpone some decisions
  - the same philosophy adopted by the late-binding mechanism.
- it depends on the awareness that a system have of itself
  - strictly related to the “self” of the object-oriented programming languages;
- it specializes some of the object-oriented basic mechanisms (constructors, invocations, and so on)
  - it exploits the classic mechanisms: inheritance, delegation;

Its use produces a better comprehension of the object-oriented mechanism and of their implementation.

---

**Referents**

- **Walter Cazzola.**
  Evaluation of Object-Oriented Reflective Models.

- **Jacques Ferber.**
  Computational Reflection in Class Based Object Oriented Languages.

- **Pattie Maes.**
  Concepts and Experiments in Computational Reflection.
  In Proceedings of OOPSLA’87, pages 147-156, Orlando, Florida, USA, October 1987. ACM.