CONCEPTS AND EXPERIMENTS IN COMPUTATIONAL REFLECTION

Pattie Maes

AI-LAB
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
pattie@arti.vub.uucp

ABSTRACT. This paper brings some perspective to various concepts in computational reflection. A definition of computational reflection is presented, the importance of computational reflection is discussed and the architecture of languages that support reflection is studied. Further, this paper presents a survey of some experiments in reflection which have been performed. Examples of existing procedural, logic-based and rule-based languages with an architecture for reflection are briefly presented. The main part of the paper describes an original experiment to introduce a reflective architecture in an object-oriented language. It stresses the contributions of this language to the field of object-oriented programming and illustrates the new programming style made possible. The examples show that a lot of programming problems that were previously handled on an ad hoc basis, can in a reflective architecture be solved more elegantly.

1. Introduction

Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation. Although "computational reflection" (further on called reflection) is a popular term these days, the issues related to it are very complex and at the moment still badly understood. The first part of the paper (sections 2 to 5) attempts to elucidate some of these issues. It presents a definition of reflection and discusses the use of reflection in programming. It further introduces the concept of a language with a reflective architecture, which is a language designed to support reflection.

This research was supported by ESPRIT project number 440: "Message Passing Architectures and Description Systems" and COST-13 project number 7: "Advanced Issues in Knowledge Representation". The author is a research assistant of the Belgian National Fund for Scientific Research.

Reflective architectures have already been realised for procedure-based (Smith,1982), logic-based (Weyhrauch, 1980) and rule-based languages (Davis,1982). The second part of the paper (sections 6 to 8) discusses the realisation of a reflective architecture in an object-oriented language (further on called OOL). Existing OOLs only support limited, ad-hoc reflective facilities, which leads to limitations and unclear designs, and consequently to problems in programming. However, over the years OOLs have evolved towards designs providing more and more reflective facilities. This paper introduces the next logical step in this evolution. It discusses an original experiment to incorporate an explicit and uniform architecture for reflection in an OOL. This experiment shows that it is possible to realise a reflective architecture in an OOL and that there are specific advantages as well to object-oriented reflection.

2. What is Reflection

This section presents a definition of computational reflection applicable to any model of computation, whether it be procedural, deductive, imperative, message-passing or other. We define **computational reflection** to be the behavior exhibited by a reflective system, where a **reflective system** is a computational system which is about itself in a causally connected way. In order to substantiate this definition, we next discuss relevant concepts such as computational system, about-ness and causal connection.

A **computational system** (further on called system) is a computer-based system whose purpose is to answer questions about and/or support actions in some domain. We say that the system is **about** its domain. It incorporates internal structures representing the domain. These structures include data representing entities and relations in the domain and a program prescribing how these data may be manipulated. Computation actually results when a processor (interpreter or CPU) is executing (part of) this program[1]. Any program

[1] In some languages the distinction between data and program is opaque. This however does not affect the understandability of the definition of reflection presented here. Also, it would be more appropriate to substitute the term "computation" by "deduction" for some languages.

that is running is an example of a computational system.

A system is said to be **causally connected** to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other. A system steering a robot-arm, for example, incorporates structures representing the position of the arm. These structures may be causally connected to the position of the robot's arm in such a way that (i) if the robot-arm is moved by some external force, the structures change accordingly and (ii) if some of the structures are changed (by computation), the robot-arm moves to the corresponding position. So a causally connected system always has an accurate representation of its domain and it may actually cause changes in this domain as mere effect of its computation.

A reflective system is a system which incorporates structures representing (aspects of) itself. We call the sum of these structures the **self-representation** of the system. This self-representation makes it possible for the system to answer questions about itself and support actions on itself. Because the self-representation is causally-connected to the aspects of the system it represents, we can say that:

(i) The system always has an accurate representation of itself.

(ii) The status and computation of the system are always in compliance with this representation. This means that a reflective system can actually bring modifications to itself by virtue of its own computation.

3. The Use of Reflection

At first sight the concept of reflection may seem a little far-fetched. Until now it has mostly been put forward as a fascinating and mysterious issue albeit without technical importance. We claim however that there is a substantial practical value to reflection. A lot of functionalities in computation require reflection. Most every-day systems exhibit besides **object-computation,** i.e. computation about their external problem domain, also many instances of **reflective computation**, i.e. computation about themselves. Examples of reflective computation are: to keep performance statistics, to keep information for debugging purposes, stepping and tracing facilities, interfacing (e.g. graphical output, mouse input), computation about which computation to pursue next (also called reasoning about control), self-optimisation, self-modification (e.g. in learning systems) and self-activation (e.g. through monitors or deamons).

Reflective computation does not directly contribute to solving problems in the external domain of the system. Instead, it contributes to the internal organisation of the system or to its interface to the external world. Its purpose is to guarantee the effective and smooth functioning of the object-computation.

Programming languages today do not fully recognise the importance of reflective computation[2]. They do not provide adequate support for its modular implementation. For example, if the programmer wants to follow temporarily the computation, e.g. during debugging, he often changes his program by adding extra statements. When finished debugging, these statements have to be removed again from the source code, often resulting in new errors. Reflective computation is so inherent in every-day computational systems that it should be supported as a fundamental tool in programming languages. The next section discusses how languages might do so.

4. What is a Reflective Architecture

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. Concretely, this means that:

(i) The interpreter of such a language has to give any system that is running access to data representing (aspects of) the system itself. Systems implemented in such a language then have the possibility to perform reflective computation by including code that prescribes how these data may be manipulated.

(ii) The interpreter also has to guarantee that the causal connection between these data and the aspects of the system they represent is fulfilled. Consequently, the modifications these systems make to their self-representation are reflected in their own status and computation.

Reflective architectures provide a fundamentally new paradigm for thinking about computational systems. In a reflective architecture, a computational system is viewed as incorporating an object part and a reflective part. The task of the object computation is to solve problems and return information about an external domain, while the task of the reflective level is to solve problems and return information about the object computation.

In a reflective architecture one can temporarily associate reflective computation with a program such that during the interpretation of this program some tracing is performed. Suppose that a session with a rule-based system has to be traced such that the sequence of rules that is applied is printed. This can be achieved in a language with a reflective architecture by stating a reflective rule such as

```
IF a rule has the highest priority in a situation,
THEN print the rule and the data which match its conditions
```

[2] Note that more advanced programming environments might provide facilities for handling some of the problems discussed here. However, typically, programming environments are not built in an "open-ended" way, which means that they only support a fixed number of those functionalities. Further, they often only support computation about computation in a static way, i.e. not at run-time.

In a rule-based language that does not incorporate a reflective architecture, the same result can only be achieved either by modifying the interpreter code (such that it prints information about the rules it applies), or by rewriting all the rules such that they print information whenever they are applied.

So clearly reflective architectures provide a means to implement reflective computation in a more modular way. As is generally known, enhanced modularity makes systems more manageable, more readable and easier to understand and modify. But these are not the only advantages of the decomposition. What is even more important is that it becomes possible to introduce abstractions which facilitate the programming of reflective computation the same way abstract control-structures such as DO and WHILE facilitate the programming of control flow.

5. Existing Reflective Architectures

Procedure-based, logic-based and rule-based languages incorporating a reflective architecture can be identified. 3-LISP (Smith, 1982) and BROWN (Friedman and Wand,1984) are two such procedural examples (variants of LISP). They introduce the concept of a reflective function, which is just like any other function, except that it specifies computation about the currently ongoing computation. Reflective functions should be viewed as local (temporary) functions running at the level of the interpreter: they manipulate data representing the code, the environment and the continuation of the current object-level computation.

FOL (Weyhrauch, 1980) and META-PROLOG (Bowen, 1986) are two examples of logic-based languages with a reflective architecture. These languages adopt the concept of a meta-theory. A meta-theory again differs from other theories (or logic programs) in that it is about the deduction of another theory, instead of about the external problem domain. Examples of predicates used in a meta-theory are "provable(Theory,Goal)", "clause(Left-hand,Right-hand)", etc.

TEIRESIAS (Davis, 1982) and SOAR (Laird, Rosenbloom and Newell, 1986) are examples of rule-based languages with a reflective architecture. They incorporate the notion of meta-rules, which are just like normal rules, except that they specify computation about the ongoing computation. The data-memory these rules operate upon contains elements such as "there-is-an-impasse-in-the-inference-process", "there-exists-a-rule-about-the-current-goal", "all-rules-mentioning-the-current-goal-have-been-fired", etc.

If we study the above mentioned reflective architectures, many common issues can be identified. One such issue is that almost all of these languages operate by means of a **meta-circular interpreter** (F.O.L. presents an exception which will be discussed later). A meta-circular interpreter is a representation of the interpretation in the language, which is also actually used to run the language[3]. Virtually, the interpretation of such a language consists of an infinite tower of circular interpreters interpreting the circular interpreter below. Technically, this infinity is realised by the presence of a second interpreter (written in another language), which is able to interpret the circular interpreter (and which should be guaranteed to generate the same behavior as the circular one).

The reason why all these architectures are this way is because a meta-circular interpreter presents an easy way to fulfill the causal connection requirement. The self-representation that is given to a system is exactly the meta-circular interpretation-process that is running the system. Since this is a procedural representation of the system, i.e. a representation of the system in terms of the program that implements the system, we say these architectures support **procedural reflection**.

The consistency between the self-representation and the system itself is automatically guaranteed because the self-representation is actually used to implement the system. So there is not really a causal connection problem. There only exists one representation which is both used to implement the system and to reason about the system. Note that a necessary condition for a meta-circular interpreter is that the language provides one common format for programs in the language and data, or more precisely, that programs can be viewed as data-structures of the language.

One problem with procedural reflection is that a self-representation has to serve two purposes. Since it serves as the data for reflective computation, it has to be designed in such a way that it provides a good basis to reason about the system. But at the same time it is used to implement the system, which means that it has to be effective and efficient. These are often contradicting requirements.

Consequently, people have been trying to develop a different type of reflective architecture in which the self-representation of the system would not be the implementation of the system. This type of architecture is said to support **declarative reflection** because it makes it possible to develop self-representations merely consisting of statements about the system. These statements could for example say that the computation of the system has to fulfill some time or space criteria. The self-representation does not have to be a complete procedural representation of the system, it is more a collection of constraints that the status and behavior of the system have to fulfill.

The causal connection requirement is more difficult to realise here: it has to be guaranteed that the explicit representation

[3] This representation minimally consists of a name for the interpreter-program (such as "eval" in LISP) plus some reified interpreter-data (such as the list-of-bindings and the continuation). It might also be richer, for example by making more explicit about the interpreter-program.

of the system and its implicitly obtained behavior are consistent with each-other. This means that in this case, the interpreter itself has do decide how the system can comply with its self-representation. So, in some sense the interpreter has to be more intelligent. It has to find ways to translate the declarative representations about the system into the interpretation-process (the procedural representation) that is implementing the system.

Such an architecture can be viewed as incorporating representations in two different formalisms of one and the same system. During computation the most appropriate representation is chosen. The implicit (procedural) representation serves the implementation of the system, while the explicit (declarative) representation serves the computation about the system. Although in architectures for declarative reflection more interesting self-representations can be developed, it is still is an open question in how far such architectures are actually technically realisable. GOLUX (Hayes, 1974) and Partial Programs (Genesereth,1987) are two attempts which are worth mentioning.

Actually the distinction between declarative reflection and procedural reflection should more be viewed as a continuum. A language like F.O.L. (Weyhrauch, 1980) is situated somewere in the middle: F.O.L. guarantees the accuracy of the self-representation by a technique called **semantic attachment**. The force of the self-representation is guaranteed by **reflection principles**. It is far less trivial to prove that the combination of these two techniques actually also succeeds in maintaining the consistency between the self-representation and the system.

6. A History of OOL with Respect to Reflection

The previous section discussed examples of existing reflective architectures in procedure-based, logic-based and rule-based languages. We now turn to object-oriented languages. Although the first OOLs, such as SIMULA (Dahl and Nygaard, 1966) or SMALLTALK-72 (Kay, 1972), did not yet incorporate facilities for reflective computation, it must be said that the concept of reflection fits most naturally in the spirit of object-oriented programming. An important issue in OOL is abstraction: an object is free to realise its role in the overall system in whatever way it wants to. Thus, it is natural to think that an object not only performs computation about its domain, but also about how it can realise this (object-) computation.

Designers of OOLs have actually felt the need to provide such facilities. Two strong motivations exist. A first motivation is the design of specialised interpreters. It seems to be very difficult to find an agreement on the fundamental principles of object-oriented programming. As it turns out the programming language community is still now actively experimenting in order to find the "basic" features an object-oriented language should support (Stefik and Bobrow, 1986): is a distinction between classes and instances necessary? what form of inheritance should be provided? what do messages look like? etc.

It has become clear that a specific design for an OOL suits some applications, but is inappropriate for others. Reflective facilities present a solution to this problem. A language with reflective facilities is **open-ended**: reflection makes it possible to make (local) specialised interpreters of the language, from within the language itself. For example, objects could be given an explicit, modifiable representation of how they are printed, or of the way they create instances. If these explicit self-representations are causally connected (i.e. if the behavior of the object is always in compliance with them) it becomes possible for an object to modify these aspects of its behavior. One object could modify the way it is printed, another object could adopt a different procedure for making instances, etc.

A second motivation is inspired by the development of **frame-based languages**, which introduced the idea to encapsulate domain-data with all sorts of reflective data and procedures (Roberts and Goldstein,1977) (Minsky,1974). An object would thus not only represent information about the thing in the domain it represents, but also about (the implementation and interpretation of) the object itself: when is it created? by whom is it created? what constraints does it have to fulfill? etc. This reflective information seems to be useful for a range of purposes:

- it helps the user cope with the complexity of a large system by providing documentation, history, and explanation facilities,
- it keeps track of relations among representations, such as consistencies, dependencies and constraints,
- it encapsulates the value of the data-item with a default-value, a form to compute it, etc,
- it guards the status and behavior of the data-item and activates procedures when specific events happen (e.g. the value becomes instantiated or changed).

OOLs have responded to this need by providing reflection in ad hoc ways. Reflective facilities were mixed in the object-level structures. In languages such as SMALLTALK-72 (Kay, 1972) and FLAVORS (Weinreb and Moon, 1981), an object not only contains information about the entity that is represented by the object, but also about the representation itself, i.e. about the object and its behavior. For example, in SMALLTALK, the class Person may contain a method to compute the age of a person as well as a method telling how a Person object should be printed. Also in FLAVORS, every flavor is given a set of methods which represent the reflective facilities a flavor can make usage of (cfr. figure 1).

```
:DESCRIBE (message): ()
GET-HANDLER-FOR: (OBJECT OPERATION)
MAKE-INSTANCE: (FLAVOR-NAME &REST INIT-OPTIONS)
:OPERATION-HANDLED-P (message): (OPERATION)
SYS:PRINT-SELF (message :PRINT-SELF):
               (OBJECT STREAM PRINT-DEPTH SLASHIFY-P)
:SEND-IF-HANDLES (message): (MESSAGE &REST ARGS)
:WHICH-OPERATIONS (message): ()
```

Fig. 1. The structure of the vanilla-flavor.

There are two problems with this way of providing reflective facilities. One is that these languages always support only a fixed set of reflective facilities. Adding a new facility means changing the interpreter itself. For example, if we want to add a reflective facility which makes it possible to specify how an object should be edited, we have to modify the language-interpreter such that it actually uses this explicit edit-method whenever the object has to be edited.

A second problem is that they mix object-level and reflective level, which may possibly lead to obscurities. For example, if we represent the concept of a book by means of an object, it may no longer be clear wether the slot with name "Author" represents the author of the book (i.e. domain data) or the author of the object (i.e. reflective data).

One step towards a cleaner handling of reflective facilities was set by the introduction of **meta-classes** by SMALLTALK-80 (Goldberg and Robson, 1983). In SMALLTALK-72 classes are not yet objects. The internal structure and message-passing behavior of an object can be specified in its class, but the structure and behavior of a class cannot be specified. The idea behind this development in SMALLTALK-80 (which was later also adopted in LOOPS (Bobrow and Stefik, 1981)) is that it should also be possible to specify the internal structure and computation of a class. Meta-classes serve this purpose.

Meta-classes already made one improvement towards the disctinction between object-information and reflective information: a meta-class only specifies system-internal information about its class (because there are no domain-data which correspond to this level). However, the confusing situation at the class-level still remained: a class in SMALLTALK-80 still mixes information about the domain and information about the implementation.

Actually one disadvantage of the introduction of meta-classes is that they introduce some confusion because the relation class/meta-class and instance/class does not run in parallel (although it is presented as if they do). As a study by Borning and O'Shea (Borning and O'Shea,1987) reveals, users of SMALLTALK are often confused with meta-classes. We suggest that this confusion might well arise because of the undisciplined split between system information and domain information. A class in SMALLTALK is sometimes viewed as an object being an instance of a meta-class (i.e. as something containing reflective information), at other times it is viewed as a class containing information about the domain (i.e. representing an abstraction).

Another step towards the origin of reflective architectures was taken by the development of OOLs such as PLASMA (Smith and Hewitt, 1975), ACTORS (Lieberman, 1981), RLL (Greiner, 1980) and OBJVLISP (Briot and Cointe, 1986). These languages try to bring more uniformity in object-oriented programming by representing everything in terms of objects. They all contribute to the uniformity of the different notions existing in OOLs by representing everything in terms of objects: class, instance, meta-class, instance-variable, method, message, environment and continuation of a message. This increased uniformity makes it possible to treat more aspects of object-oriented systems as data for reflective computation.

In general, it can be said that the evolution of OOLs tends towards a broader use of reflective facilities. In the beginning reflective facilities were only used in minor ways. A class would for example only represent the reflective information telling what its instances were. However, as OOLs evolved, the self-representations became richer and applied in a broader way (from instances only, to classes, to meta-classes, to messages, etc).

However none of the existing languages has ever actually recognised reflection as the primary programming concept developers of OOL are (unconsciously) looking for. The languages mentioned above only support a finite set of reflective facilities, often designed and implemented in an ad hoc way. The next section discusses in what ways an OOL with a reflective architecture differs from these languages. It highlights the issues that were missing in the existing languages.

7. A Reflective Architecture in an OOL

This section discusses an OOL with an architecture for procedural reflection. The discussion is based on a concrete experiment that was performed to introduce a reflective architecture in the language KRS (Steels,1986). The resulting language is called 3-KRS (Maes,1987). The important innovation of 3-KRS is that it fulfills the following crucial properties of an object-oriented reflective architecture[4]:

1. A first property is that it presents the first OOL adopting a **disciplined split** between object-level and reflective level. Every object in the language is given a **meta-object**. A meta-object also has a pointer to its object. The structures contained in an object exclusively represent information about the domain entity that is represented by the object. The structures contained in the meta-object of the object hold all the

[4] None of the languages discussed above fulfills the entire list, although they might fulfill one or more of the properties.

reflective information that is available about the object. The meta-object holds information about the implementation and interpretation of the object (cfr. figure 2). It incorporates for example methods specifying how the object inherits information, how the object is printed, how a new instance of the object is made, etc.

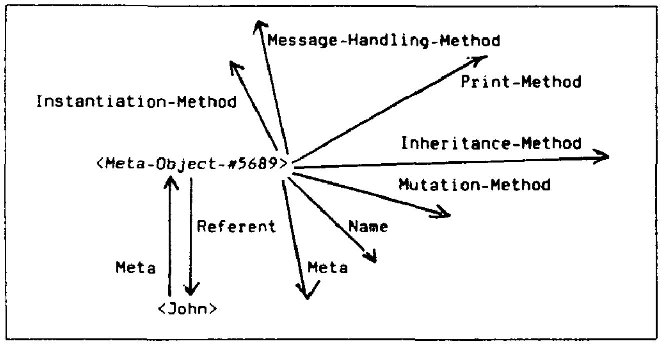


Fig. 2. An object and its meta-object.

Note that the meta-relation is not collapsed with the instance-relation (as it is in SMALLTALK-80 or LOOPS). The object John has a type-link to the Person object and a meta-link to its meta-object (named "Meta-Object-#5689").[5]

Note also that although there is a one-to-one relation between objects and meta-objects (which might suggest to combine them into one object), it is important that object and meta-object are also physically separated (which is again not true for the meta-classes of SMALLTALK). This way a standard message protocol can be developed between an object and its meta-object. This protocol makes it possible to create abstractions of the behavior of an object (i.e. ready-made meta-objects), and to temporarily attach such a special behavior to an object.

2. A second property is that the self-representation of an object-oriented system is **uniform**. Every entity in a 3-KRS system is an object: instances, classes, slots, methods, meta-objects, messages, etc. Consequently every aspect of a 3-KRS system can be reflected upon. All these objects have meta-objects which represent the self-representation corresponding to that object. Note that since meta-objects are again objects, meta-objects have to be created in a lazy way. KRS incorporates a lazy-construction mechanism which takes care of this (Van Marcke,1986): meta-objects are only constructed when they are actually needed.

3. A third property is that 3-KRS provides a **complete self-representation**. The meta-objects contain all the information about objects that is available in the 3-KRS language. Actually, the contents of meta-objects was designed on the basis of the interpreter. The code of the interpreter was divided in blocks which represent how a specific aspect of a certain type of object is implemented. All of these blocks were afterwards reified (i.e. made explicit) under the form of objects (fillers of slots in the meta-objects). 3-KRS incorporates a set of primitive meta-objects which together represent the complete 3-KRS interpreter (cfr. figure 3). When a specific object is created in some application, it will automatically inherit one of these meta-objects from its type.

[5] However the "meta" slot of an object is also inherited. When the object John does not override the "meta" slot, it will when needed make a copy of the meta-object of Person.

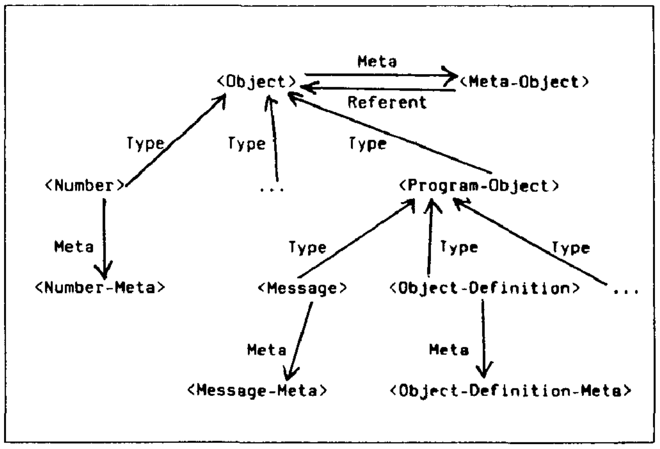


Fig. 3. The primitive meta-objects of 3-KRS, or the theory the language 3-KRS incorporates about the implementation of its objects and the interpretation of its programs.

"Meta-Object" is the most general meta-object. It roughly contains what was illustrated in figure 2. The other meta-objects in the figure above add to or specialise the information in Meta-Object. For example, Message-Meta represents the information that is available about message-objects. It adds to Meta-Object slots representing the method to be used to evaluate the message and the continuation and environment of the evaluation.

4. A fourth property is that the self-representation of a 3-KRS system is **consistent**. The self-representation is actually used to implement the system. The explicit representation of the interpreter that is embedded in the meta-objects is used to implement the system. Whenever some operation has to be performed on an object, e.g. an instance of the object has to be created or the object has to answer a message, or a message-object has to be evaluated, the meta-object of the object is requested to perform the action. The technique that is used in order to avoid an infinite loop is that there is a second, implicit interpreter which is used to implement the default (or standard) behavior[6].

5. A final property is that the self-representation can also at

[6] The real (i.e. implicit) interpreter of the 3-KRS language tests for every operation that it has to perform on an object whether the meta-object of this object specifies a deviating method for this operation. "Deviating" meaning here: different from (overriding) the methods of the primitive meta-objects listed in figure 3. If so, the interpreter will apply the explicit method (3-KRS program). If not, it handles this operation implicitly. This implicit handling guarantees the same results as the explicit methods described in the primitive meta-objects.

run-time be modified, and these modifications actually have an impact on the run-time computation. The self-representation of the system is explicit, i.e. it consists of objects. Thus, any computation may access this self-representation and make modifications to it. These modifications will result in actual modifications of the behavior of the system.

The 3-KRS experiment is extensively described in (Maes, 1987). It shows that it is feasible to build a reflective architecture in an object-oriented language and that there are even specific advantages to object-oriented reflection. These advantages are a result of the encapsulation and abstraction facilities provided by object-oriented languages. The next section illustrates these advantages. It presents two examples of programming in an object-oriented reflective architecture.

8. A New Programming Style

Although the implementation of 3-KRS is far from trivial, from the programmer's point of view the language has a simple and elegant design. The basic unit of information in the system is the object. An object groups information about the entity in the domain it represents. Every object in 3-KRS has a meta-object. The meta-object of an object groups information about the implementation and interpretation of the object. An object may at any point interrupt its object-computation, reflect on itself (as represented in its meta-object) and modify its future behavior.

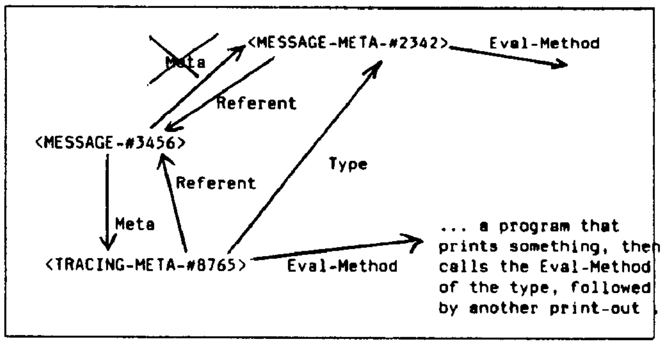
Reflective computation may be guided by the object itself or by the interpreter. An object may cause reflective computation by specifying reflective code, i.e. code that mentions its meta-object. The interpreter causes reflective computation for an object whenever the interpreter has to perform an operation on the object and the object has a special meta-object. At that moment the interpretation of the object is delegated to this special meta-object.

This reflective architecture supports the modular construction of reflective programs. The abstraction and encapsulation facilities inherent to OOLs make it possible to program object-computation (objects) and reflective computation (meta-objects) independently of each other. There is a standard message protocol between an object and its meta-object which guarantees that the two modules will also be able to work with each other[7]. This makes it possible to temporarily associate a certain reflective computation with an object without having to change the object itself. Another advantage is that libraries of reflective computation can be constructed.

This section (schematically) illustrates what programming in a reflective OOL is like. It demonstrates the particular style of modular programming that is supported by reflective architectures. More (operational code) examples of programming in 3-KRS can be found in (Maes,1987).

A first example illustrates the object-oriented equivalent of the tracing example presented in section 4. The reflective architecture of 3-KRS provides a modular solution for implementing reflective computation such as stepping and tracing of programs. One can temporarily associate a meta-object with a program (-object) such that during its evaluation various tracing or stepping utilities are performed. Note that the object itself remains unchanged, only its meta-object is temporarily specialised to a meta-object adapted to stepping or tracing.

Figure 4 illustrates the idea. Message-#3456 is an object representing some message. It has a meta-object, called Message-Meta-#2342 which may be a copy of the default meta-object for a message or a user-defined specialisation of this. The Tracer-Meta object is designed to be temporarily attached to any program-object. The meta-link from the program-object to the old meta-object is temporarily replaced by a meta-link to (a copy of) the Tracer-Meta. Tracer-Meta-#8765 inherits from this old meta-object and overrides the Eval-Method: it adds some actions before and after the eval-method of the old meta-object (such that the evaluation itself is still handled by Message-Meta-#2342). These actions will take care that when Message-#3456 is evaluated, some information is printed before and after the evaluation.



<MESSAGE-META-#2342>
Eval-Method
Meta
Referent
<MESSAGE-#3456>
Type
Referent
Meta
<TRACING-META-#8765>
Eval-Method
... a program that prints something, then calls the Eval-Method of the type, followed by another print-out

Fig. 4. Associating a tracing behavior temporarily.

Notice that it is not only possible to add before or after methods. The eval-method itself could also be overridden or specialised (it is again an object that can be manipulated).

A second example illustrates how a local deviating interpreter may be realised. A major advantage of a language with a reflective architecture is that it is open-ended, i.e. that it can be adapted to user-specific needs. But even more, a reflective architecture makes it possible to dynamically build and change interpreters from within the language itself. It allows for example to extend the language with meaningful constructs without stepping outside the interpreter. Note that this way the language itself can be made more concise (and thus more efficient). The extra structure and computation

[7] More specifically, the meta-object has to specify values for a predefined set of slots (variables and methods), which for the 3-KRS experiment roughly correspond to the names listed in figure 2. Actually this set varies according to the type of object at hand. E.g. the meta-object of a program-object in addition has to specify an evaluation-method.

necessary to provide objects with special features such as documentation, constraints or attachment do not have to be supported for all objects in the system but can be provided on a local basis.

Figure 5 illustrates a very simple example. The 3-KRS language does not support multiple-inheritance. However, if a multiple-inheritance behavior is needed for some object (or class of objects), it can be realised by a specialised meta-object. The object Mickey-Mouse has a deviating interpreter which takes care of the multiple-sources inheritance behavior of this object. The specific strategy for the search of inherited information is implemented explicitly in the language itself by overriding the inheritance-method of the default meta-object.

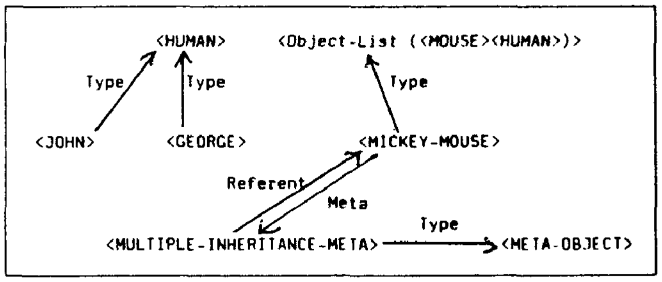


Fig. 5. Implementing a local variation on the language.

For frequently used variations on the language, abstractions may be provided. The 3-KRS system currently provides an initial library of reflective behaviors including meta-objects for pretty-printing, tracing and stepping, several variations on the language (multiple-inheritance, frames, monitors, streams, defaults, etc). The programmer can simply pick such a special behavior and attach it to an object in his application. Very few slots of such a meta-object remain to be filled.

Note finally that the architecture of object-oriented reflection provides a sophisticated control of the granularity of reflective computation. Local reflective computation can be obtained by making reflective individual instances. E.g. a reflective object John, or a reflective particular message. More general reflective computation can be obtained by making reflective abstract objects (which serve as the type of other objects). E.g. one can make all person objects reflective, by making the class person object reflective. Or one can make a class of messages in the system behave in a special way, by making their class message object reflective.

9. Discussion and Conclusions

We can conclude that the experiment of 3-KRS does for the object-oriented paradigm what languages like 3-LISP, F.O.L. and TEIRESIAS did for the procedure, logic and rule-based paradigm respectively. Just like these languages, 3-KRS introduced a new concept (or programming-construct) being the notion of a meta-object. Meta-objects are just like the other objects of the language, except that they represent information about the computation performed by other objects and that they are also taken into account by the interpreter of the language when running a system.

Another common issue is the way the causal connection requirement is handled. Just like the main part of the languages discussed in section 5, 3-KRS represents an architecture for procedural reflection. 3-KRS is run by a meta-circular interpreter: the self-representation that is given to a system is an explicit representation of the implementation of the system. Consequently this self-representation also represents the system in terms of the concepts inherent in the interpretation of an object-oriented language: handling messages, creating instances, etc.

This paper briefly introduced some of the concepts and experiments in computational reflection. However, many aspects of reflection, reflective architectures and particularly of object-oriented reflection (its implementation and use) have not been discussed in this paper. The interested reader may consult (Maes,1987).

10. Acknowledgements

I am very grateful to Luc Steels, who supervised this research and when necessary corrected its direction. I also would like to thank Pierre Cointe for the valuable comments he provided.

11. Bibliography

Bobrow D. and Stefik M. (1981) "The LOOPS manual". Tech. Rep. KB-VLSI-81-13. Knowledge Systems Area. Xerox Palo Alto Research Center. Palo Alto, California.

Borning A. and O'Shea T. (1987) "Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltak-80 Language". In: *Proceedings of the ECOOP Conference*. Paris, France.

Bowen K. (1986) "Meta-level Techniques in Logic Programming". In: *Proceedings of the International Conference on Artificial Intelligence and its Applications*. Singapore.

Briot J.P. and Cointe P. (1986) "The OBJVLISP Model: Definition of a Uniform Reflexive and Extensible Object-Oriented Language". In: *Proceedings of the European Conference on Artificial Intelligence - 1986.*

Dahl O. and Nygaard K. (1966) "SIMULA - An Algol-Based Simulation-Language". *Communications of the ACM.* 9: 671-678.

Davis R. (1982) In: "Knowledge-Based Systems in Artificial Intelligence". Davis R. and Lenat D. Mc Graw-Hill, New York.

Friedman D. and Wand M. (1984) "Reification: Reflection without meta-physics". *Communications of the ACM.* Vol 8.

Genesereth M. (1987) "Prescriptive Introspection". In: *Meta-Level Architectures and Reflection.* Eds: P. Maes and D. Nardi. North-Holland, Amsterdam, June 1987.

Goldberg A. and Kay A. (1976) "SMALLTALK-72 Instruction Manual". Technical Report SSL-76-6, Xerox Palo Alto Research Center. Palo Alto, California.

Goldberg A. and Robson D. (1983) "Smalltalk-80: The Language and its Implementation". Addison-Wesley. Reading, Massachusetts.

Greiner R. (1980) "RLL-1: A Representation Language Language". Stanford Heuristic Programming Project. HPP-80-9. Stanford, California.

Hayes P. (1974) "The Language GOLUX". University of Essex Report. Essex, United Kingdom.

Laird J., Rosenbloom P. and Newell A. (1986) "Chunking in SOAR: The Anatomy of a General Learning Mechanism". In: *Machine Intelligence.* Vol 1. Nr 1. Kluwer Academic Publishers.

Lieberman H. (1981) "A Preview of ACT1". Massachusetts Institute of Technology, Artificial Intelligence Laboratory. MIT AI-MEMO 625. Cambridge, Massachusetts.

Maes P. (1987) "Computational Reflection". PhD. Thesis. Laboratory for Artificial Intelligence, Vrije Universiteit Brussel. Brussels, Belgium. January 1987.

Minsky M. (1974) "A Framework for Representing Knowledge". Massachusetts Institute of Technology, Artificial Intelligence Laboratory. MIT AI-MEMO 306. Cambridge, Massachusetts.

Roberts R. and Goldstein I. (1977) "The FRL Primer". Massachusetts Institute of Technology, Artificial Intelligence Laboratory. MIT AI-MEMO 408. Cambridge, Massachusetts.

Smith B. (1982) "Reflection and Semantics in a Procedural Language". Massachusetts Institute of Technology. Laboratory for Computer Science. Technical Report 272. Cambridge, Massachusetts.

Smith B. and Hewitt C. (1975) "A PLASMA Primer (draft)". Massachusetts Institute of Technology. Artificial Intelligence Laboratory. Cambridge, Massachusetts.

Steels L. (1986) "The KRS Concept System". Vrije Universiteit Brussel. Artificial Intelligence Laboratory. Technical Report 86-1. Brussels, Belgium.

Stefik M. and Bobrow D. (1986) "Object-Oriented Programming: Themes and Variations". In: *AI magazine.* Vol. 6. No. 4.

Van Marcke K. (1986) "A Parallel Algorithm for Consistency Maintenance in Knowledge Representation". In: *Proceedings of the European Conference on Artificial Intelligence, 1986.* Brighton, England.

Weinreb D. and Moon D. (1981) "Lisp Machine Manual". Symbolics Inc. Cambridge, Massachusetts.

Weyhrauch R. (1980) "Prolegomena to a Theory of Mechanized Formal Reasoning". In: *Artificial Intelligence* Vol. 13 No. 1,2. North Holland. Amsterdam. The Netherlands.