VRIJE UNIVERSITEIT BRUSSEL

ARTIFICIAL INTELLIGENCE LABORATORY

Pattie Maes

COMPUTATIONAL REFLECTION

technical report 87_2

Pattie Maes

COMPUTATIONAL REFLECTION

technical report 87_2

[illegible]

[illegible]

[illegible]

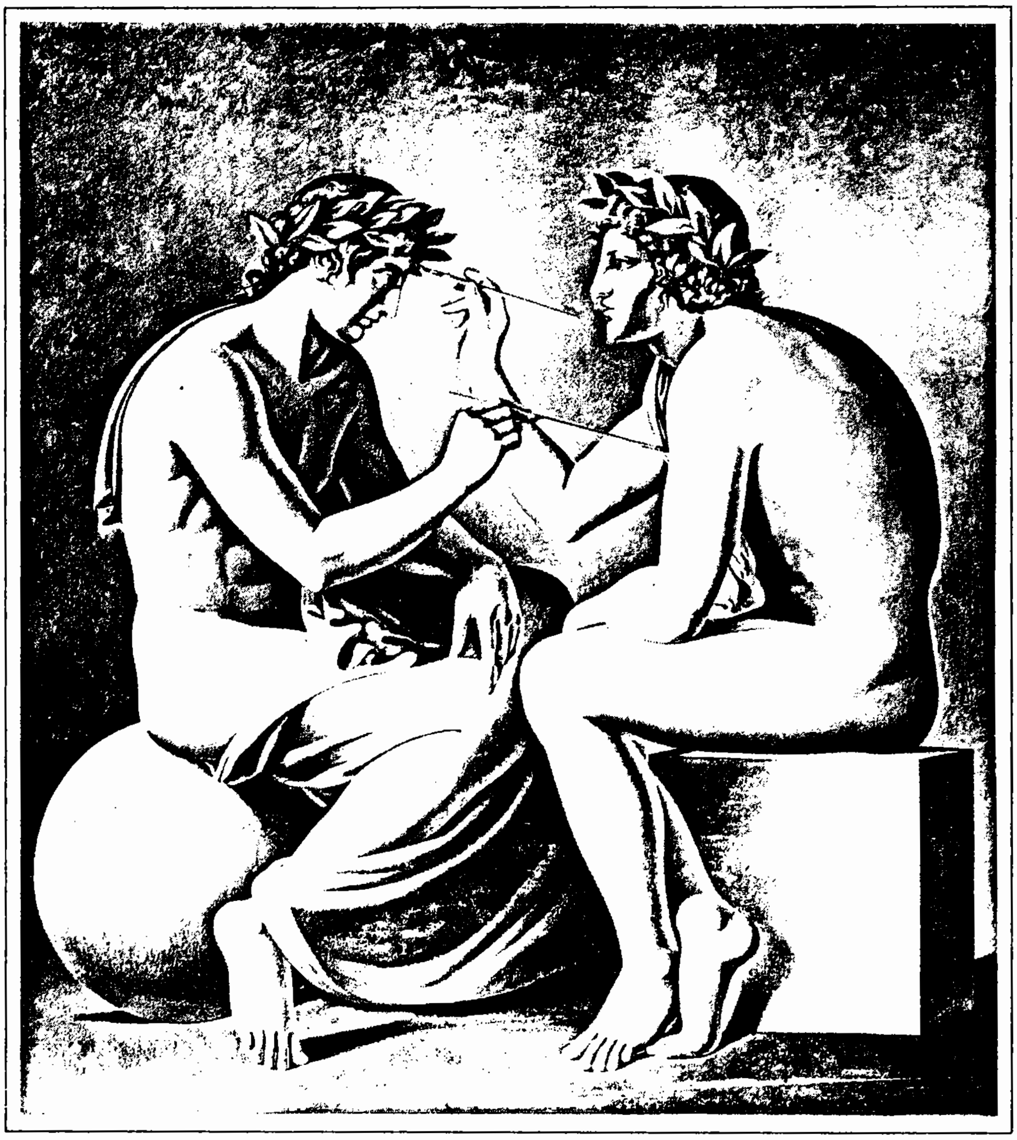
for my parents

Abstract

Computational reflection is a new approach for introducing modularity in programs. This work makes two original contributions towards a better understanding and utilisation of this approach. The more general contribution is that it brings some perspective to various issues in computational reflection. A definition of computational reflection is presented, the importance of computational reflection is discussed and the design and construction of architectures that support reflection is studied. Illustrations are drawn from different sorts of languages including procedural, logic-based and rule-based languages. The more specific contribution is that it presents the first effort to introduce reflection in an object-oriented language. The implementation of a concrete reflective architecture is worked out and the programming style made possible by this architecture is extensively illustrated. The examples show that a lot of programming problems that were previously handled on an ad hoc basis, can in an architecture for computational reflection be solved more elegantly.

Keywords:

Artificial Intelligence, Introspection, Meta-Architecture, Meta-Knowledge, Object-Oriented Language, Programming Language, Reflection, Reflective Architecture, Self-Reference.



Carlo Maria Mariani

La Mano Ubbidisce all'Intelletto, (1983)

Summary Table of Contents

Table of Contents

Acknowledgements

First of all I would like to thank my supervisor Luc Steels for teaching me that research is a playful and fascinating activity. I would also like to thank him for creating the material and intellectual environment that made this work possible.

Apart from Luc Steels, some other people substantially improved the presentation of this work. Walter Van de Velde, Kris Van Marcke and Ken Haase all read this text very carefully and provided very useful comments. I also would like to thank the members of my thesis committee, Pierre Cointe, Michel Sintzoff, Theo D'Hondt and Anton Nijholt for the valuable suggestions they provided.

The members of the VUB AI-LAB provided a stimulating and cordial environment. The people I share my office with, Viviane Jonckers, Kris Van Marcke and Walter Van de Velde were always willing to discuss open questions or solve technical problems.

I also profited from a most interesting stay at the Xerox Palo Alto Research Labs, which gave me the opportunity to personally interact with many of the leading researchers in this field.

Finally, I would like to thank the Belgian National Fund for Scientific Research whose fellowship I enjoyed the last three and a half years. The research presented in this work was also partially supported by ESPRIT project 440 "Message Passing Architectures and Description Systems" and the COST-13 project on "Advanced Issues in Knowledge Representation".

Introduction

1. Sketch of the Problem

Most computational systems exhibit besides object-computation, i.e. computation about their problem domain, also reflective computation, i.e. computation about their own computation. 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 (in the case of non-deterministic programs), 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 domain of the computational 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. This work argues that reflective computation is so inherent in real world computational systems that it should be supported as a fundamental tool in programming languages.

Programming languages today do not fully recognise the importance of reflective computation. 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 degugging, these statements have to be removed again from the source code, often resulting in new errors.

Some types of reflective computation might be supported by a programming environment for the language. Nevertheless, in general the reflective computation of systems has to be programmed in ad hoc ways and reflective code has to be mixed into object-level code. Often programmers are forced to re-implement reflective computation over and over again for every new application.

2. The Proposed Solution

Important advances in programming are often the result of the introduction of larger modular structures. Consequently, the evolution of programming languages is characterised by a search for more modularity. We believe that explicit support for reflective computation is such a step towards more modularity.

Reflective architectures are architectures which provide tools for handling reflective computation explicitly. First of all they support the decomposition of computation in separate modules for object-computation and reflective computation. They also provide the tools to program these types of computation independently and to assemble them later without too many interaction problems.

The decomposition of computation into object-computation and reflective computation introduces more modularity into computational systems. As is generally known, enhanced modularity makes computational 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 computational 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.

A computational system implemented using a reflective architecture has data representing its own object-computation. The system is able to access and manipulate those data at run-time. These data are causally connected to the object-computation itself, which means that they are always a faithful representation of the computation and that changes to the data are also reflected in the actual object-computation of the system. A reflective architecture should support frequently used operations on those data.

For example, in a computational system simulating a chemical process, the code which describes the computation that has to be pursued is intertwined with code that specifies how statistics of the simulation should be kept, how the graphical output should be constructed, how a backtrace of the course of the process is built up, etc. This simulator would in a reflective architecture consist of object-level modules which contain code specific to the simulation domain, and reflective modules which contain code concerned with this computation.

There would for example be a module for keeping statistics, one for handling the graphical output, one that builds up a backtrace of the course of the process, one that organises the data-directed style of computation, etc. The two types of modules are linked to each other, such that the data of the reflective modules are causally connected to the appropriate entities in the computation of the object-level modules.

The run-time computation of the simulation system would then take place at two levels. The computation at the object-level takes care that the simulation is performed. It manipulates data representing the problem domain. The computation at the reflective level takes care of the internal organisation of the computational system and its interface to the outside world. It manipulates data representing the actual object-level computation that is going on.

3. Contributions

The original contributions made in this work are of two types. A first type is of a more general nature. Although many people, especially those from the Artificial Intelligence tradition, have intuitively felt the importance of reflection, particularly for building intelligent systems (Hofstadter,1981) (Doyle,1980) (Perlis,1985), the issue has until now mostly been covered by studies of a more philosophical or a more theoretical nature. We believe that progress in this area can only be made through solid technical work.

The issues related to the computational aspect of reflection, i.e. how reflective systems can be built, are very complex and at the moment still badly understood. The first international workshop on computational reflection took place only recently (Maes and Nardi,1987). This work presents a first attempt to make a complete, technical study of the issues in computation reflection. It presents a definition of computational reflection, identifies the notion of a reflective architecture, discusses what to do and what not to do with reflection, studies design issues of reflective architectures, including the components and possibilities, and the relation between function and structure.

The second type of contribution is of a more specific nature. Although a reflective architecture has already been realised for procedural, logic-based and rule-based languages, this work presents the first realisation of a reflective architecture in an object-oriented language. It extensively describes how a reflective architecture was realised in the language KRS by means of accessible object-oriented circular interpreters. This experiment shows that the well-known advantages of an object-oriented language have also a substantial value for reflective architectures, and that object-oriented languages may also gain from this confrontation.

4. Overview

Chapter I presents a definition of computational reflection. It defines concepts such as computational systems, meta-systems and reflective systems. It also discusses languages which provide facilities for implementing reflective computation. It shows that there exist a lot of languages with limited, compiled-in reflective facilities, but only few that provide an explicit and uniform architecture for reflection. The latter type of languages will be called languages with a reflective architecture.

Chapter II shows that reflection is a fundamental concept in real-world programming that should consequently be supported by any high-level programming language. It presents concrete examples of programming problems which would in languages with reflective facilities be greatly facilitated.

Chapter III presents examples of existing languages which were designed for supporting reflective computation. The design and use of a procedural, a logic-based and a rule-based language with a reflective architecture are presented. The chapter argues that these languages support a new form of modular programming.

Chapter IV identifies the common problems that have to be faced when building a language with a reflective architecture. It also discusses the techniques that can be used to solve these problems.

Chapter V focuses on the relation between the structure of a reflective architecture and the reflective fuctionality that results. It argues that the reflective architectures that have already been built have only explored a fraction of the designs that can be adopted.

Chapter VI illustrates issues in the implementation of a reflective architecture with a concrete example. It shows how a reflective architecture can be built into an object-oriented language. The discussion describes a concrete experiment that was performed to realise a reflective architecture in the language KRS (Steels,1986). The chapter also surveys the facilities for reflection provided by existing object-oriented languages and stresses the innovations of this new language.

Chapter VII illustrates by means of the same experiment what programming in a reflective architecture is like. It shows how a lot of programming problems that were previously supported on ad hoc basis can by means of such a reflective architecture be solved more elegantly. Examples presented include interfacing, tracing and metering, variations on the language, and self-optimisation.

We conclude with a summary of the achievements and some topics for further research.

CHAPTER I

What is Reflection

1. Introduction

Although "reflection" is a popular term these days, it does not yet have a well-established meaning and is certainly not used in a uniform way. The purpose of this chapter is to introduce a definition for this concept and thus lay a foundation for further discussions. We will define a reflective system as a computational system that reasons and acts upon itself. In order to substantiate this definition we will discuss relevant concepts such as computational system, meta-system and causal connection. At the end of the chapter the notion of a reflective architecture is introduced.

2. Definitions

2.1. Computational Systems

A **computational system** (here called a system) is something that **reasons** about and **acts** upon some part of the world, called the **domain** of the system (cf. figure 1). A computational system represents its domain under the form of **data**. Its **program** prescribes how these data should be manipulated. It dictates the actions that can or must be taken in order to reason about and act upon the domain in a way that complies with the semantics of the domain, i.e. with the relations and properties of entities that hold in the domain.

A program is made up of lines of code (also called sentences) in some programming language. We say that the computational system is **implemented** in this language. The **computation** of the system actually results because of the execution of the program by an **execution process**. This is either an interpreter for the programming language in which the program is written or the CPU for programs written in machine language. The computation returns results that convey new information about the domain or that actually act upon the domain.

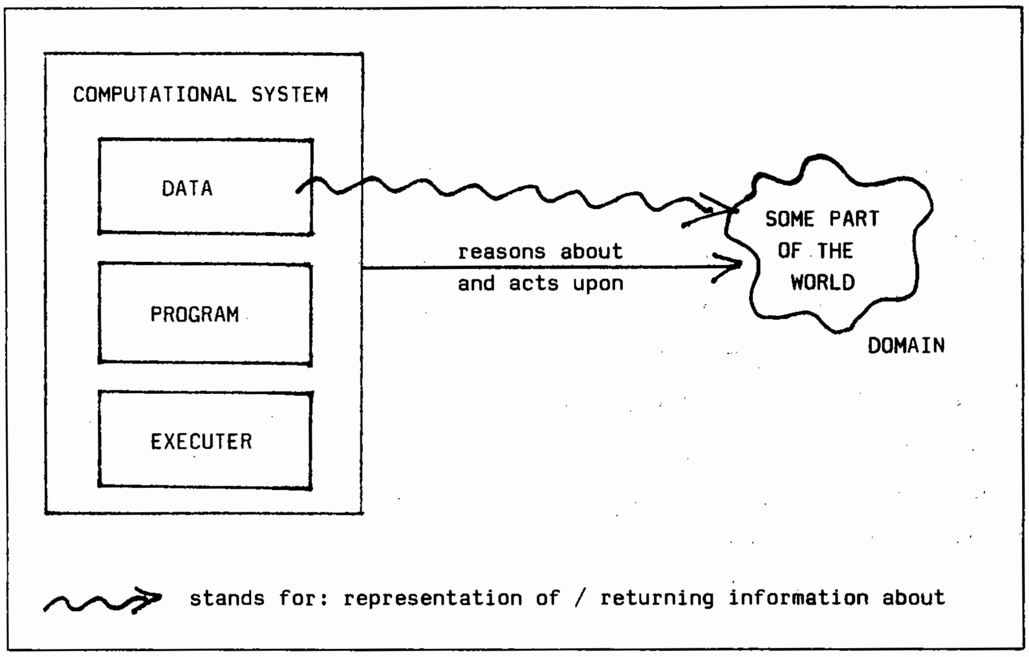


Fig. 1. A computational system.

An accounting system, for example, is a computational system which has as domain the world of accounts and financial transactions. Its data are representations of the actual accounts of some organization. Its program specifies how these data should be manipulated such that valid information can be returned about the domain or appropriate actions can be taken in the domain. It specifies for example, that in a transfer between accounts, the amount that is added to one account should be the same as the amount that is subtracted from the other account.

The accounting system reasons about its domain when answering questions about financial transactions and the state of various accounts. It acts upon its domain by creating new accounts and by making transfers between accounts. Its results convey new information about the financial state of the organization. Suppose this system is implemented in PASCAL, then the program of the accounting system are sentences in the PASCAL language and the executer of the system is a PASCAL compiler/executer.

2.2. Causal Connection

Most of the time, the links between the domain of a computational system and the computational system itself are taken care of by hand. The programmer is responsible for making the data and program of the system be an honest representation of the domain. And the user is responsible for the appropriate interpretation of the results returned by the computation into the actual domain. For example, an accounting system is built and also used in a way that guarantees that its data and programs mirror the actual financial situation of the organization.

But a computational system may also be **causally connected** to its domain. This means that the system and its domain are linked in such a way that if one of the two changes, this leads to an effect upon the other[1]. A system steering a robot-arm, for example, incorporates data representing the position of the arm. These data are 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 data change accordingly (e.g. through sensors) and (ii) if some of the data are changed, the robot-arm moves to the corresponding position.

When implementing a causally connected system, the programmer only has to set up the causal connection link once. After that, the representation and return relations between the system and its domain are automatically guaranteed. So a causally connected system and its domain can, through the causal connection link, actually act upon one another without human interaction.

Since the program of a computational system represents the semantics of the domain, and since these semantics are most of the time fixed, the problem of causal connection between them does not really exist. Consequently, the causal connection link only has to be implemented between the domain and the data of the system. Under this condition, a system is said to be causally connected to its problem domain if (cf. figure 2)

(i) when data of the system change, the entities in the domain represented by these data are affected, and

(ii) when entities in the domain change, the data representing these (or aspects of these) in the system are affected.

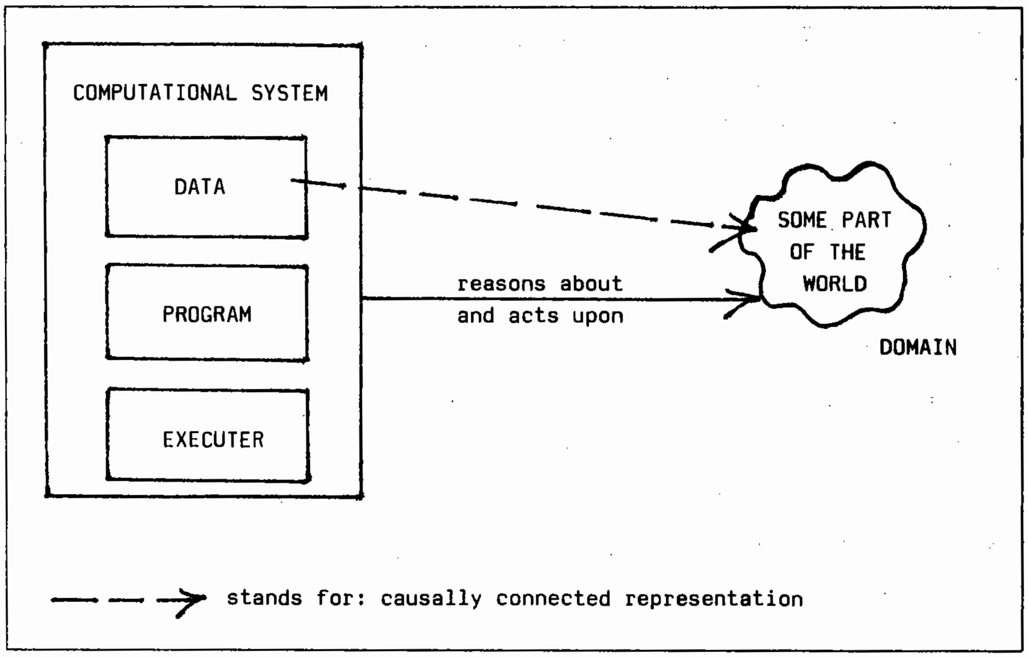

COMPUTATIONAL SYSTEM
DATA
PROGRAM
EXECUTER
reasons about and acts upon
SOME PART OF THE WORLD
DOMAIN
stands for: causally connected representation

Fig. 2. A causally connected computational system.

Implementing a causal connection between the data and the domain of a system involves specifying what the data of the system ought to represent. For example, if the data of the system should represent true properties of entities in the domain, then the causal connection link should be constructed such that it guarantees that this consistency relation holds at all times. Which is to say that when a change happens in either the data or the domain, the effects that are caused in the other are such that the data never attribute false nor unknown properties to the entities of the domain.

For most computational systems, the programmer is not too much concerned with specifying this representation relation, because it is the programmer/user himself who "implements" this relation: he has to decide to what properties and relations of entities in the real world the data of the system correspond. For causally connected systems, it is crucial to formally specify which data in the system should represent what aspects of the entities in the domain, such that some mechanism can try to maintain these specifications.

The concepts of representation and causal connection will prove to be very important for meta and reflective systems.

2.3. Meta-systems

A **meta-system** is a computational system that has as its domain another computational system, called its **object-system**. Thus a meta-system is a system that reasons about and acts upon another computational system. A meta-system has a representation of its object-system in its data. Its program specifies **meta-computation** about the object-system and is therefore called a **meta-program**. The meta-computation returns new information about the object-system or actually acts upon the object-system (cf. fig 3).

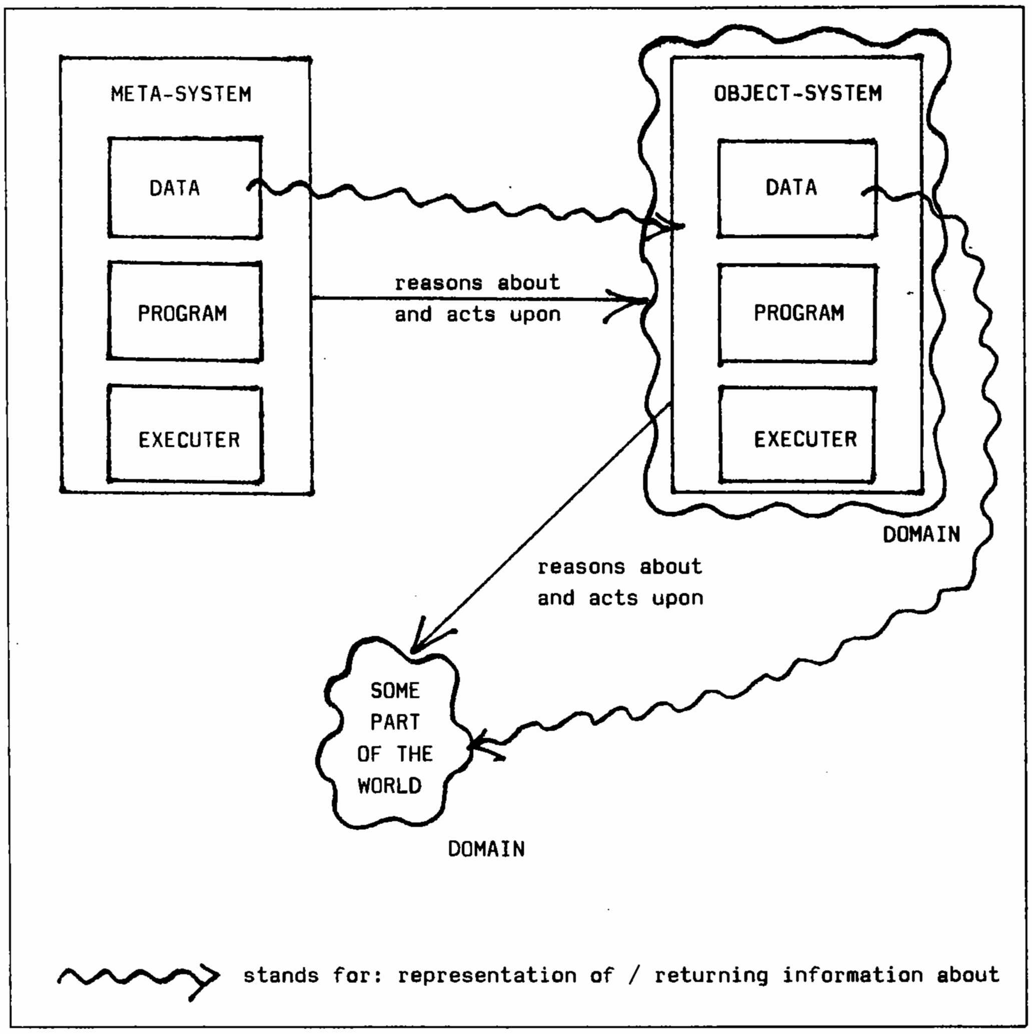


Fig. 3. A meta-system.

(Hayes-Roth,1983) defines "meta-X" as "X about X", meaning X about something of the same type as X. For example, a **meta-system** is a system about another system. A **meta-program** is a program about another program. **Meta-computation** is computation about other computation. The X that fills the about-role in these relations is called the **object-X**. Consequently we also use the terms **object-system, object-program** and **object-computation** in the context of meta-systems.

A programming language interpreter is a well-understood example of a meta-system. An interpreter for LISP, for example, is a system that reasons and acts upon systems implemented in LISP. When the interpreter is running, i.e. interpreting a LISP program, it does meta-computation, or computation about computation. In particular, it emulates the computation of its object-system by virtue of its own computation.

The data of the LISP-interpreter are representations of the object-system's program, data and state of computation. They incorporate for example variables bound to the sub-expression that is currently being evaluated, the stack, the environment of interpretation, etc. The program of the interpreter prescribes how to execute LISP systems: it tells how to simulate the LISP object-system in compliance with the semantics of the LISP language.

There may be a causal connection link between the data of the meta-system and what they represent, namely entities and relations in the object-system. A run-time program-optimiser and a debugging system are examples of causally connected meta-systems. They make modifications to their object-system in such a way that the behavior of this object-system is also actually affected. The causal connection link is here established by modifying the code of the object-system the way the meta-computation dictates. The object-system can only change by modifications prescribed by the meta-level.

Another example of a causally connected meta-system is that of an operating system. An operating system has a lot of object-systems, called processes. It has representations of these processes. It keeps information about them such as the time they were created, their current status, the computer time already used, their priority, etc. The operating system reasons about these processes when deciding which one to activate or to kill. It also acts upon its object-systems by actually activating or killing them, which means by starting up or terminating their computation. The causal connection link is realised here because this meta-system actually executes its object systems.

2.4. Reflective Systems

Reflection is the process of reasoning about and/or acting upon oneself. A reflective system is a causally connected meta-system that has as object-system itself[2]. The data of a reflective system contain, besides the representation of some part of the external world, also a causally connected representation of itself, called the **self-representation** of the system (cf. figure 4).

The program of a reflective system prescribes, besides the computation about a domain, also **reflective computation** about itself. It specifies how the data representing the system itself should be manipulated in compliance with its semantics. So, the program of a reflective system actually defines part of the semantics of the system.

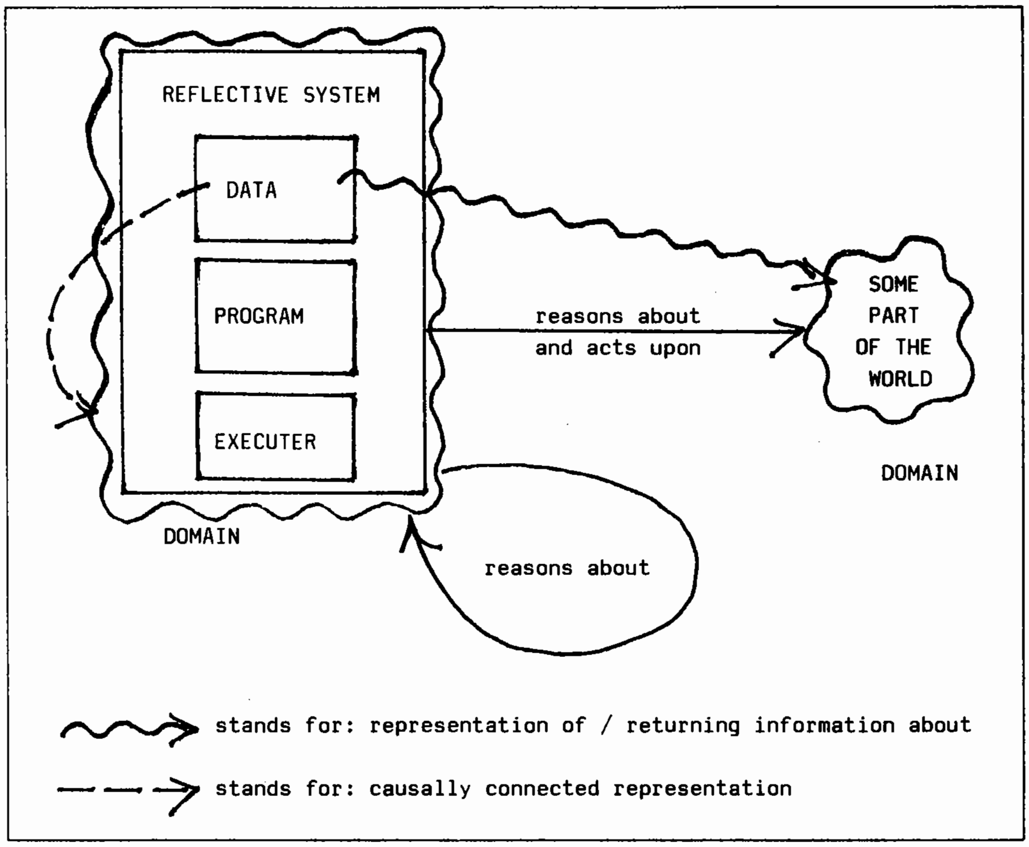

REFLECTIVE SYSTEM
DATA
PROGRAM
EXECUTER
DOMAIN
reasons about and acts upon
SOME PART OF THE WORLD
DOMAIN
reasons about
stands for: representation of / returning information about
stands for: causally connected representation

Fig. 4. A reflective system.

When a system is reasoning about or acting upon itself, we speak of **reflective computation**. The code in the program specifying such reflective computation is called **reflective code**.

The object- and reflective code necessarily have the same format (are in the same language), because there is only one executing process which produces both sorts of computation. This means that the same representation and reasoning capabilities are used for object-computation and reflective computation. In contrast, meta-systems are often implemented in a different language than their object-systems.

The following subsections present a few simple examples of reflective computation. More examples are given in chapter II.

2.4.1. Example 1

LISP supports various reflective functions. Some of these reflective functions can actually modify programs by means of destructive operations. This makes it possible to compute and modify the program of a computational system to be executed at run-time. Consider for example the following definitions[3]

```
(define integer-stream ()
   '(0))

(define integer-stream-element ()
   (let* ((list-of-integers (integer-stream))
          (last-generated-integer (car list-of-integers)))
      (eval '(define integer-stream ()
                ',(cons (1+ last-generated-integer)
                        list-of-integers)))
      last-generated-integer))
```

Fig. 5. Modifying the program to be executed at run-time in LISP (in italics).

Integer-stream computes the stream of integers. Integer-stream-element returns the next integer of the integer-stream. It also destructively modifies the definition of the function integer-stream. The first time integer-stream-element is called, it returns 0

```
(integer-stream-element) --> 0
```

and redefines the function integer-stream to be

```
(define integer-stream () '(1 0))
```

so that the next time integer-stream-element is called, it returns 1

```
(integer-stream-element) --> 1
```

The function integer-stream is again redefined to be

```
(define integer-stream () '(2 1 0))
```

and so on.

2.4.2. Example 2

Another field in which samples of reflective computation frequently recur, is expert systems. The need is often felt in expert systems to reason about the knowledge and computation of the system itself during computation. The system EMYCIN provides several handles to alter the computation from within an application system itself. Consider the following example from MYCIN (Van Melle,1980):

```
(a) IF it is not known whether there are factors that interfere with
       the patients bleeding,
    THEN it is definite (1.0) that there are no factors that interfere
         with the patients bleeding.
```

Fig. 6. Reflective computation in EMYCIN.

Reflective computation is used here to deal with the problem of incomplete knowledge. Rule (a) shows a special type of rule, called a self-referencing rule. This rule refers to the own status of the system. It is used to form a conclusion out of uncertainty. When all the regular rules have failed to make a conclusion about whether there are interfering factors or not, rule (a) will conclude that there are no interfering factors.

2.4.3. Example 3

An example of a reflective system stemming from yet another background is a learning system. A system with learning capabilities reasons about itself and acts upon itself in such a way that its performance at a certain task improves. The system LEX2 discussed in (Mitchell,1983), for example, has a representation of its own program which it can access and modify. The purpose of LEX2 is to become better at solving integrals.

The system incorporates a program (in the form of a set of rules) for each type of integral. For example (from Mitchell,1983)

```
S: ∫ 3x cos(x) dx --> Apply substitution
                        with u = 3x, and dv = cos(x) dx

G: ∫ f1(x) f2(x) dx --> Apply substitution
                        with u = f1(x), and dv = f2(x) dx
```

Fig. 7. Part of the LEX2 program.

Initially solving an integral with this set of rules involves a lot of search. But after such a search, the LEX2 system analyses the search tree to see whether the rules cannot be refined to avoid search in the future. E.g. after some positive and negative training instances, LEX 2 has modified the above program to

```
S: ∫ 3x trig(x) dx --> Apply substitution
                         with u = 3x, and dv = trig(x) dx

G1: ∫ poly (x) f2(x) dx --> Apply substitution
                              with u = poly(x), and dv = f2(x) dx

G2: ∫ f1(x) transc(x) dx --> Apply substitution
                               with u = f1(x), and dv = transc(x) dx
```

Fig. 8. The revised LEX2 program (after reflection).

This is only possible because the LEX2 system has a representation of its own program that it can analyse and make changes to.

2.5. Meta-Programming Versus Reflective Programming

The causal connection requirement in the definition of reflective systems is crucial to distinguish between systems that only support meta-programming, and systems that actually support reflection (which entails meta-programming). A lot of systems represent aspects of computational systems without causally connecting these representations to the system itself. Suppose we have a system implemented in PROLOG, in which we define a predicate "Demo" which represents the top-level of the PROLOG interpreter (cf. figure 9).

```
demo(true).
demo(A,B) :- demo(A), demo(B).
demo(X) :- clause(X,Y), demo(Y).
```

Fig. 9. Top-level loop of a circular PROLOG interpreter.

Symbols in this program starting with a capital represent variables. Symbols starting with a lowercase character represent constants. The ":-" sign indicates (reverse) logical implication, while commas should be read as "and". The first clause states that the goal "true" is always proven. The second clause states that the conjunction of two goals is proved if you prove both of them. The third clause states that a goal can be proved by finding a clause in which this goal is on the left-hand side of a clause, and by then proving the right-hand side of this clause. Note that the first and third clause together make that all clauses of the form

```
p(T1,...,TN) :- true.
```

also written as

```
p(T1,...,TN).
```

are proved.

The system in which this predicate is incorporated can be said to have data representing its own interpreter. Since in PROLOG one can also view programs as data the computational system is even able to simulate the interpretation of programs and predict the outcome. Moreover, the system is able to do some limited reasoning about its own interpretation. E.g. it can make use of clauses such as

```
innocent(X) :- not(demo(guilty(X))).
```

However, recall that the representation of the interpreter should be causally connected to the real interpretation if we want the system to be able to make correct predictions (i.e. to have an accurate representation of the interpreter). The representation of the PROLOG interpreter in the above example is not linked to the real PROLOG interpreter. It might be a faulty or incomplete representation of the interpreter. It does not represent the CUT operator for example. Because there is no causal connection, the system is not at all able to affect the real underlying interpreter-process by making changes to the representation of the interpreter.

We say that such systems incorporate **disconnected** (Smith,1986) representations of (aspects of) themselves. If one changes the representation of the system, the system itself is not affected by it. Vice versa, if the system itself changes, the representation does not reflect this change. So it is due to the causal connection link that a reflective system is able to actually reason about and act upon its own behavior.

3. Languages with Reflective Facilities

Several approaches can be identified for building reflective systems. They can be classified according to the paradigm that is used for combining object-computation and reflective computation.

First, there are systems that incorporate some form of reflection in an ad hoc way. For example, learning systems necessarily exhibit some sort of reflective behavior since they are able to make improvements to themselves. However most learning systems today, implement this reflective ability in a ad hocs way. The design choices, such as what situations trigger the reflective computation (i.e. learning), what representations are manipulated during reflective computation, or how these manipulations affect the future performance of the system, are not made explicit. This makes it difficult to compare different designs or experiment with alternatives.

A second group of systems makes use of reflective facilities provided by the programming language in which they are implemented. Some programming languages incorporate a selection of meta-constructs to support specific types of reflective computation. LISP is an example of such a language. LISP provides functions for reflecting upon programs given as data (e.g. eval, apply), for reflecting upon the run-time stack (e.g. catch & throw) and for reflecting upon the run-time environment (e.g. boundp). These functions give LISP-systems the possibility to access and manipulate parts of their own computation[4].

Systems implemented in such languages have only limited possibilities for reflection, because the set of meta-constructs is not open-ended. The different types of reflective computation which can be implemented are fixed by the design of the language. Another limitation of these languages is that reflective computation is not full-fledged computation. E.g. LISP provides functions to access and manipulate the run-time stack, but in most LISPs this stack is not a real data-object which can be manipulated in the way object-data are manipulated. We can for example not name a stack and pass it as an argument to a call of the evaluator.

A third category of reflective systems are those which are implemented in a programming language that provides a uniform and open-ended architecture for the specification of reflective computation. Examples of such languages are 3-LISP (Smith,1982), FOL (Weyhrauch,1980) and TEIRESIAS (Davis,1982). These languages allow all systems implemented in them to reflect upon themselves in an explicit and uniform way. All systems have access to and can modify a causally connected representation of themselves. We will say that these languages have a **reflective architecture**. A characteristic of languages with a reflective architecture is that object programming is identical to reflective programming: they have the same representational and computational behavior with **linking rules** or **reflection principles** between the two levels to guarantee the causal connection link. Because object computation and reflective computation are identical, reflection can recur.

4. Languages with a Reflective Architecture

The purpose of a programming language is to provide building blocks for the implementation of computational systems. One building block consists of a syntactical construct in the language and the appropriate machinery in its interpreter to make this building block work in the overall language.

A programming language stresses the importance of a **programming concept** by providing building blocks for it and supporting (or even forcing) the use of these blocks as much as possible. Some languages, e.g. machine language, are designed to support very simple and small blocks as the unit of construction. Other languages, e.g. higher level languages, provide larger and more complex units.

We say that a language has an **architecture for X** if the language stresses the importance of the programming concept X and thus supports building blocks for X. For example, a programming language such as MODULA, has an architecture for modules because it stresses the importance of modules in programming. It provides building blocks for arranging the program of a computational system in modules. A programming language, such as SMALLTALK, has an architecture for data-abstraction because it provides syntactic constructs and the associated interpretative machinery for realising data-abstraction.

A programming language is said to provide an architecture for reflection, or also to have a **reflective architecture**, if it recognises reflection as a fundamental programming concept and thus provides building blocks for reflection and encourages their use as much as possible. This implies that the language provides syntactical constructs for specifying reflective code and that the interpreter of the language is able to actually execute this reflective code during computation.

So a language with a reflective architecture supports (besides object-computation) the explicit and uniform representation of the reflective computation of the computational systems that are implemented in it. Programs in such a language have the possibility to specify reflective computation. They can employ syntactical constructs for specifying reflective data and reflective code. The language interpreter guarantees that the causal connection requirement between these data and the aspects of the system they represent is fulfilled. In this way the resulting computational systems have access to causally connected representations of themselves during computation. The modifications they make to these representations during computation are reflected in their own behavior.

Note that a lot of confusion with the term "reflection" originates from a failure to respect the distinction between languages with a reflective architecture and reflective systems. Reflective systems are systems that actually show a reflective behavior. While a language with a reflective architecture has an architecture that supports the construction of potentially reflective systems: systems implemented in it have the capability for reflective computation.

5. Reflective Architectures Versus Meta-level Architectures

On many points the concept of a reflective architecture runs parallel to the concept of a meta-level architecture. There is often confusion between these two types of architectures. Therefore it is interesting to work out their exact differences.

A programming environment has a **meta-level architecture** if it has an architecture which supports meta-computation, without supporting reflective computation. A meta-level architecture is designed for building systems which are about other systems. However, it does not make it possible to build systems which are about themselves.

We purposely use the term programming environment here, instead of programming language, because typically meta-level programming environments present different languages for the object-computation and the meta-computation[5]. These two languages each have their own syntax and interpreter.

A programming environment in which the code of the interpreter is made accessible and modifiable to the programmers, is an example of a meta-level architecture. For example, assume an environment for PROLOG programming in which the source of the PROLOG interpreter is a program written in LISP, made accessible to the programmer. The programmer can make variants of the PROLOG interpreter, by making modifications to this LISP program and recompiling it. The environment may even provide especially designed tools for specialising the interpreter to be used for a particular PROLOG program. Figure 10 illustrates this idea.

```
(define interpret-PROLOG-program (program goals)
   (let ((special-interpreter
            (get 'special-interpreter
                 (program-name program))))
      (funcall (or special-interpreter 'default-PROLOG-interpreter)
               program goals)))

(define default-PROLOG-interpreter (program goals)
   ....)

(define modify-interpreter-for-program (program-name new-interpreter)
   (putprop 'special-interpreter new-interpreter program-name))

(define tracing-information-generating-interpreter (program goals)
   ...)

(define reduced-language-interpreter (program goals)
   ...)

(define extended-language-interpreter (program goals)
   ...)
```

Fig. 10. Allowing program-dependent variations on the interpreter.

The function default-PROLOG-interpreter represents the default interpreter for PROLOG programs. It takes as input a file containing the source-code PROLOG program and the list of goals we want to prove with this program. The function modify-interpreter-for-program allows the specification of a special interpreter for a specific PROLOG program. For example:

```
(modify-interpreter-for-program
     'my-program
     'tracing-information-generating-interpreter)
```

The above expression specialises the interpreter to be used for the program with name "my-program". The function tracing-information-generating-interpreter could for example print information about the status of the interpretation.

When the LISP program above is interpreting a PROLOG program, it is a meta-system for this PROLOG system. The interpreter has a representation of its object-system (the program to run). It incorporates variables bound to the program-source, the goals to be proved, the specific interpreter to be used for this program, etc.

However, notice that this representation of the object-system is only modifiable from the meta-level. The object-system itself has no access to this representation of itself. This implies that modifications to this meta-representation can only happen before the interpretation of the PROLOG program starts. A programmer can modify the interpreter of a program before running it, but the program itself can not modify its own interpreter during computation. Figure 11 gives an illustration.

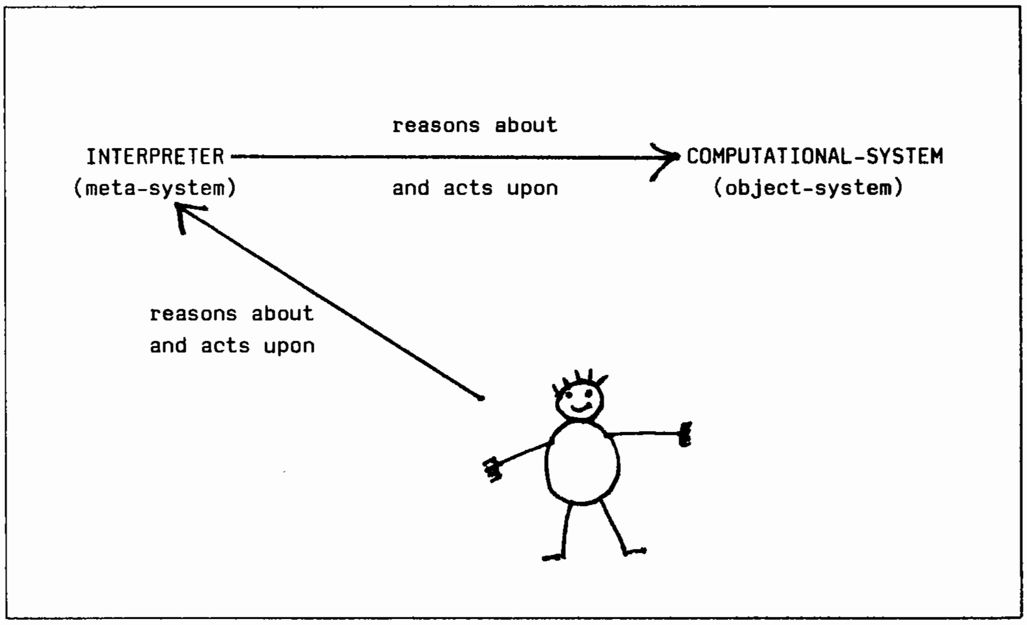


Fig. 11. A meta-interpreter.

Many concrete examples of such programming environments can be cited. For example, nearly all compilers are able to compile themselves. Another popular example is the notion of meta-interpreters and partial evaluation, as used in the LISP and logic programming communities. A lot of PROLOG dialects, for example, provide a facility for changing the interpreter to be used for some program. They incorporate a default-meta-interpreter which can be used as a basis for building a special-purpose meta-interpreter. The obtained variation of the default-meta-interpreter can be compiled together with the program that it is designed for, thereby producing a more efficient special-purpose interpreter.

Contrary to the above example, meta-interpreters are written in the same language as the programs they interpret. However, this is the only difference that exists between the notion of a meta-interpreter and the example presented above. So, although the language of the object and meta-level are the same, meta-interpreters only provide **static access** to the interpretation of a program. Meta-interpreters are mainly used to implement variants on the language. We find examples in the literature of enhanced meta-interpreters to implement different control strategies (Gallaire and Lasserre,1982), to implement deduction with uncertainties (Shapiro,1983b), to implement analysis and debugging tools (Shapiro,1983a), etc.

The alternative example, illustrating a reflective architecture, is the notion of a meta-circular interpreter, as for example used in 3-LISP. 3-LISP provides facilities for **dynamically** accessing and changing the interpreter for a program from within that program itself. The example in section 2 of chapter III demonstrates these facilities. Figure 12 presents an illustration[6].

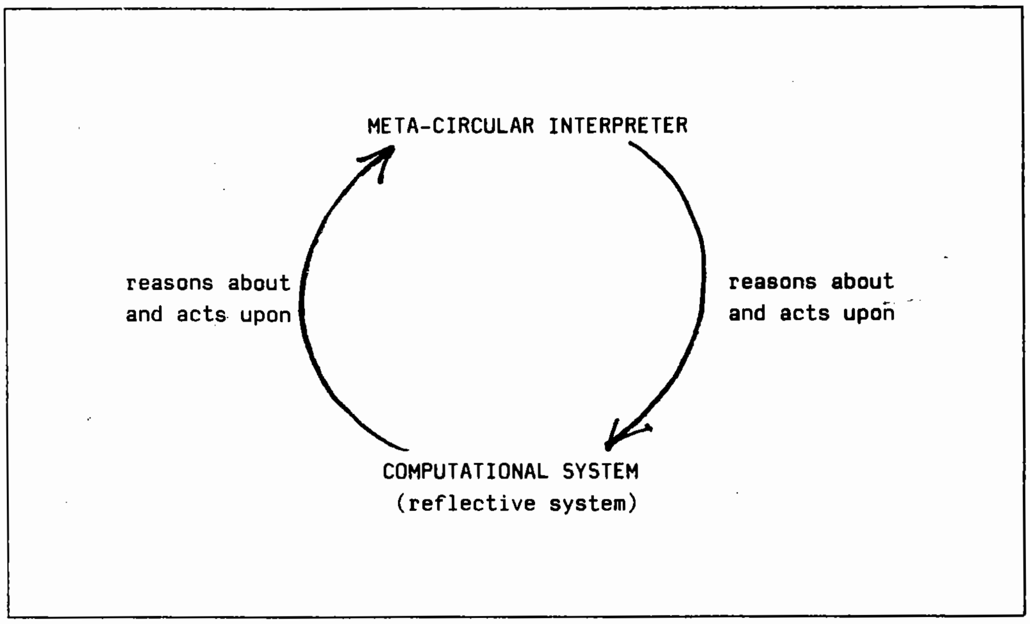

META-CIRCULAR INTERPRETER
reasons about and acts upon
reasons about and acts upon
COMPUTATIONAL SYSTEM (reflective system)

Fig. 12. A meta-circular interpreter.

Here the meta-level is accessible from the object-level at run-time. Consequently, the possibilities for meta-computation are of a different order. The program itself is able to interrupt its computation, change something to its interpretation and continue with a modified interpretation process afterwards. It is for example possible in 3-LISP to specify how the interpretation should proceed form a particular error, and to activate this piece of code whenever an error in the computation occurs. This can be realised because in 3-LISP, the code to be executed to process a program is (i) computed at run-time and (ii) accessible and modifiable from the object-level.

These functionalities can clearly not be realised by means of meta-interpreters. The special-purpose meta-interpreter for a PROLOG program is partially evaluated together with the program before running the program. Partial evaluation is a program transformation technique which happens at compile-time. The purpose of partial evaluation is to optimise the program and its interpreter as much as possible at compile-time. This means that a lot of the information on the program and its interpreter gets lost. E.g. the conditionals which can be evaluated at compile-time are no longer explicit in the code that is run. So the code to be executed to proccess a program is (i) computed at compile-time and (ii) thus not accessible and modifiable from the object-level.

(Maes,1986d) presents a more extensive overview of the differences between meta-level architectures and reflective architectures.

6. Conclusions

A computational system is said to **reason about** some part of the world if it incorporates data representing this part of the world, which it can manipulate during computation.

A computational system is said to also **act upon** a part of the world if the data representing the part of the world are **causally connected** representations, which means that the data and the part of the world they represent are linked in such a way that a change to one of them affects the other.

A computational system is called a **meta-system** if it reasons (and possibly also acts upon) another computational system, which is to say that it incorporates data representing this other computational system (possibly in a causally connected way).

A computational system is said to be **reflective** if it incorporates and also manipulates causally connected data representing (aspects of) itself.

A language with a **reflective architecture** is a language in which all systems have access to a causally connected representation of themselves.

A programming environment has a **meta-level architecture** if it has an architecture which supports meta-computation, without supporting reflective computation.

The main difference between a meta-level architecture and a reflective architecture is that a meta-level architecture only provides **static access** to the representation of a computational system, while a reflective architecture also provides **dynamic access** to this representation. The result is that a meta-level architecture is more efficient, but less general than a reflective architecture.

The next chapter presents an initial identification of the kind of programming problems reflective architectures are good for. The chapter ends with a section on the dangers of reflection, or what reflection should not be used for.

NOTES

[1] The term "causal connection" was introduced by B. C. Smith in the context of reflective systems (Smith,1982). We believe it can be applied to (and is relevant for) computational systems in general.

[2] Since recently, B. Smith adopts a different definition for reflection (Smith,1986). What we call reflection here, he now calls introspection. Smith would say that an introspective system is also actually reflective if it has a representation of (and is thus also able to reason about and act upon) itself relative to its embedding world. For example, if it has a reflective representation about whether its data are an honest representation of the domain or not. Another example would be that the system can represent at a reflective level that the semantics exhibited by its program is inconsistent with the real semantics of the domain. This notion of reflection has sprung out of the idea of the **circumstantial relativity** of language and thought. According to (Smith,1986) a representational structure not only has

> (i) a **content**, the thing the structure refers to in a particular use of the structure (depending on the set of circumstances of that particular use),
>
> (ii) and a **meaning**, indicating what and how this representational structure contributes to the content of any larger structure in which it participates (something the structure has on its own),

But a representational structure also has, and this is new,

> (iii) a **significance**, including not only the content of a particular use, but the full conceptual and functional role that the representational structure plays in and for the system that incorporates it.

Content and meaning of a structural representation are specifiable independent of conceptual and functional role. But, a great deal of the significance of a representational structure is not directly or explicitly represented by any of the structures by which it is composed. Instead it is often relative to its circumstances. Systems which are able to reason about and act upon themselves in more than just a local way, which are able to "derelativize" their thoughts and actions of the "here, now and self", are called reflective systems according to Smiths definition. No concrete reflective systems have already been built.

[3] The integer-stream is not a realistic example in the sense that a more obvious solution would be to use closures or #' & Rplaca. The Eval & Define combination was chosen to clearly show the novice how in LISP one may at run-time (i.e. from within the language itself) define and manipulate programs and then call the interpreter on them.

[4] Stepping and tracing facilities are often provided by the programming environment of a language. We claim that the problems of stepping and tracing are basically not different from other reflective programming problems such as statistics and interfacing. These types of computation should in a consistent programming language design be handled in a similar way. This becomes possible in reflective architectures.

[5] Some reflective facilities are only available for interpreted programs (and not for compiled programs).

[6] Some authors call a system where meta-level and object-level have their own special-purpose language a **two-level architecture** as opposed to a meta-level architecture.

[7] It is interesting to compare the notion of reflection with that of recursion. Non-reflective computation is realised by a known number of computational levels. For example, my PROLOG program is interpreted by an interpreter written in LISP, which is processed by machine language. Reflection means that computation is recursively defined by a beforehand unknown number of computational levels. The number of executer levels which is needed in order to realise the computation can only be decided at run-time. The recursion in the computation of the code to be executed stops at the level where no more reflective functionalities are used. Just like recursion, reflection allows you to define in a very concise manner very powerful (meta-) computation.

CHAPTER II

Uses of Reflection

1. Introduction

This chapter argues that a lot of functionalities in computation require reflection. As the examples will show, having access to representations of aspects of the system itself during computation is often very useful. The purpose of the discussion is to demonstrate that reflective computation should be supported by programming languages.

A systematic overview of the positive uses of reflection is difficult to present. However, it is possible to catalogue these uses on the basis of the aspects of a system they reason about and act upon. Following the definition of a computational system presented in chapter I, aspects of a computational system that can be made accessible for reflective computation (cf. figure 1) are:

(i) The program of the system,

(ii) The data of the system,

(iii) The program of the executing process,

(iv) The data of the executing process,

and so forth (the program and data of the process executing the executing process).

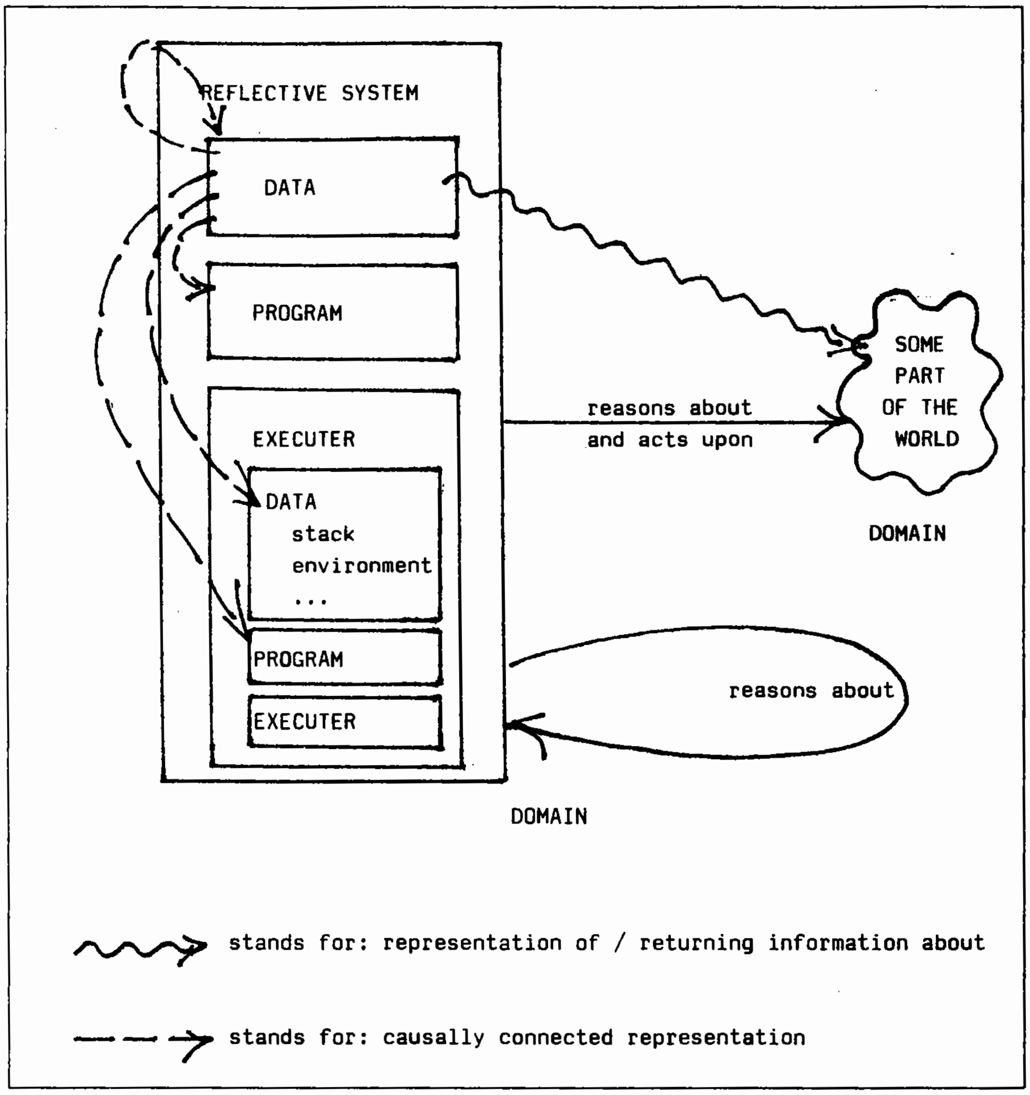


Fig. 1. Some possible self-representations.

It is easy to identify several interesting reflective uses of all these sorts of self-representations: (i) makes it possible to influence the code of a computational system to be executed at run-time. (ii) stands for data about the data of a computational system. This type of self-representation is useful for the maintenance, acquisition and communication of data. (iii) makes it possible to implement variants or extensions on the language. (iv) opens a way to alter the environment of the execution process of a computational system at run-time. The most straightforward use is the manipulation of the flow of control from within the computation itself.

This chapter does not present a complete overview of the uses of these different types of self-representations (and their combinations). Instead it presents a number of typical real-world programming problems which involve the use of these four types of self-representations.

Note that more advanced programming environments might provide facilities for handling some of the problems discussed. 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 (cf. chapter III). One of the interesting perspectives of reflection is that it makes it possible to construct and use a programming environment from within the language itself. Which has of course the advantages of readability, open-endedness and portability.

2. Reflective Computation about ones Program: Computation about the Problem of Control

The primary reason why a lot of programming languages incorporate reflective constructs, is that reflection makes it possible to control the flow of computation in a flexible way. Giving computational systems causally connected access to a representation of their own program introduces very powerful forms of control. It allows systems to decide about what to do next, not only on the basis of data about the problem domain, but also on the basis of data about their own program.

The problem of controlling computation differs according to the model of computation that is adopted in a programming language. In the **deterministic model** of computation, adopted by programming languages such as LISP or PASCAL, the program of a computational system exactly prescribes the sequence of steps a system has to take. The control aspect of the system is fixed by the program of the system. The deterministic model is suited for implementing the type of systems for which the flow of control can be defined before the system is executed. Consequently, the problem of flexible control is merely a problem of handling the exceptions in which the predefined flow of control does not work. This use of reflection is discussed in section 4.

This section discusses the problem of control in the **non-deterministic model**[1] of computation, adopted in languages such as PROLOG or OPS5 (Brownstow, Farell, Kant and Martin,1985). In the non-deterministic model, the sequence of steps that has to be taken during computation is not known beforehand. The program of a computational system describes a search-space of possible paths that could be followed during computation. However, it does not define how this search space should be explored. Computation is viewed as a search for some desired state in this search space.

The search space of real-world problems is typically very large, such that exhaustive search (trying all of the possible paths systematically) quickly leads to combinatorial explosions. Consequently, a primary concern of research in the non-deterministic model is to control this search, i.e. to find the most effective ways to explore this search space. Progressively more powerful strategies to do so have been developed. When first non-deterministic languages were developed, researchers tried to develop general, domain-independent strategies to control the search. Examples of **general strategies** are depth-first search or hill climbing (Nilson,1971). These strategies were hard-wired in the interpreter of a language.

Later, it became clear that domain-independent strategies would not suffice. The expert systems wave introduced the idea of knowledge-based or **domain-specific strategies** to control the search. Concretely, this domain-specific knowledge was implemented by guiding the search by means of heuristic rules.

More recently, a lot of non-deterministic languages also incorporate techniques to decide upon the flow of computation from within the computation itself. While building real-world systems, programmers felt the need to incorporate **context-specific strategies** in systems. MYCIN for example incorporates several reflective functionalities to reason about the flow of computation from within the computation: prevention of circular reasoning, antecedent rules, the preview mechanism, the concept of a unity path, the initial-data mechanism, self-referencing rules, etc (cf. Van Melle,1980). However, these functionalities are wired into the inference engine of MYCIN in an ad hoc way.

Consequently, the idea has arisen to design non-deterministic languages that allow a computational system to specify computation about the use of its own program. The system TEIRESIAS (Davis,1982) was one of the earliest rule-based languages to provide an architecture for reasoning about the rules of a system, as well as for using those rules to reason about the problem domain. Reflective computation in TEIRESIAS has the same status as object-level computation. Reflective computation is specified by means of reflective rules, or rules which are about other rules. The ad hoc selection of reflective facilities of MYCIN can be implemented in a system like TEIRESIAS in a much more elegant way.[2]

Consider for example the self-referencing rules of MYCIN discussed in chapter I. In principle, the order in which rules are executed in MYCIN is arbitrary. However, in order to be able to deal with self-referencing rules, the MYCIN interpreter incorporates an obscure piece of code, which ensures that self-referencing rules are only applied after the regular rules that can possibly make a conclusion about the goal. Figure 2 shows how control over the use of self-referencing rules can be implemented in TEIRESIAS (Davis,1982).

```
IF 1) there are rules which do not mention the current goal
      in their premise, and
   2) there are rules which do mention the current goal in
      their premise

THEN it is definite that the former should be used before the latter
```

Fig. 2. The control on the use of self-referencing rules is made explicit in TEIRESIAS.

This rule makes explicit both to the system and to the programmer what is happening. In addition, possibilities for reflective computation about rules have become open-ended in TEIRESIAS. The TEIRESIAS system provides (i) a meta-language which can be used to create reflective rules and (ii) a mechanism to refer to object-rules from within a reflective rule. Using these facilities, many control strategies can be realised in TEIRESIAS without being possible in MYCIN (Davis and Buchanan,1984).

Even more, because of its reflective facilities, TEIRESIAS makes it possible to define run-time control-strategies of the three types discussed previously in this section. General control strategies are implemented by means of reflective rules which make reference to all the rules in the rule-base. For example, the following reflective rules implement a forward reasoning strategy

```
IF 1) there are rules whose premises are matched
      by the current data, and
   2) there are rules whose premises are not matched
      by the current data,

THEN it is definite (1.0) that the former should be used
     before the latter.
```

Domain-specific control strategies are implemented by means of reflective rules that make reference to the set of rules that mention a particular domain problem. For example, the following reflective rule is extracted from TEIRESIAS' application domain

```
IF 1) if the infection is pelvic-abscess, and
   2) there are rules which mention in their premise
      enterobacteriaceae, and
   3) there are rules which mention in their premise grampos-rods,

THEN there is suggestive evidence (.4) that the former should be
     done before the latter.
```

Context-specific control strategies are implemented by means of reflective rules that make reference to the current status of the computation. For example, the following reflective rule invokes a procedure to handle situations in which the computation cannot proceed.

```
IF all the rules that mention the current goal in their
   conclusion have failed to achieve the goal,

THEN the rule mentioning a no-change impasse in its premise
     is definitely useful (1.0)
```

These examples show that reflection presents an ideal foundation for handling the problem of controlling non-deterministic computation.

3. Reflective Computation about ones Data: Maintenance, Acquisition and Communication of Information

The maintenance, acquisition and communication of the information contained in a computational system is a knowledge-intensive task[3]. The knowledge required for this task was previously recorded in informal and unorganised ways. It was scattered in manuals, lines of code, programmers and users notes, or was sometimes not recorded at all. In addition, it was represented in different formats and conventions.

Frame-based languages (Minsky,1974)(Roberts and Goldstein,1977) have introduced the idea to represent this knowledge explicit as normal data within the computational system itself. A data-item in a frame-based system is surrounded by a whole set of reflective data and procedures. These are used for different purposes:

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

Not only does the knowledge for the maintenance, acquisistion and communication of the information contained by a system become represented in a uniform format, but it also becomes explicit, which means that the system itself can also make use of it.

Figure 3 gives an example. It represents the general form and a specific instance of a frame. The item "age of John" in the knowledge base contains a lot of data which convey internal system information.

```
ITEM a name
   value: the current value of this item
   source-of-current-value: user-supplied/default/computed/...
   modifiable: yes/no
   showable: yes/no
   documentation: a string documenting this data item
   part-of: the more global item this item is part of
   author: the name of the author of this data item
   when-created: the date this item was created
   when-last-accessed: the date the value of this item was last accessed
   constraints: a list of constraints the value has to fulfill
   type: the type the value has to fulfill
   comparison-function: the method that should be used to compare the
                        value of this item with another value
   default-value: the value to be used when no value is stored and no
                  value can be derived or computed
   items-depending-on-this-value: the list of information items that
                                  have made use of this value to compute
                                  their own value
   items-this-value-depends-on: the list of information items whose
                                value was used to compute this value

ITEM age of John
   value: 27
   source-of-current-value: computed
   modifiable: yes
   showable: yes
   documentation: "this item represents the age of the object John"
   part-of: John
   author: pattie
   when-created: 3/10/76
   when-last-accessed: 7/12/85
   constraints: (and (< 0 value) (> 100 value))
   type: integer
   comparison-method: equal
   default-value: 25
   items-depending-on-this-value: (adultp-of-John)
   items-this-value-depends-on: (birthyear-of-John current-year)
```

Fig. 3. A frame includes reflective data about a data-item.

Apart from the slot "value", all the slots of an item represent reflective data about this item. These data are frequently used during update and retrieval of the value of the item. For example, when the item "age of John" is asked for its value, the interpreter checks whether this item is showable to the outside world (the salary of John would not be showable). When somebody wants to set the value of the age of John, the interpreter checks whether this item is modifiable, whether the proposed value is of the appropriate type and whether this value fulfills the constraints. Subsequently, it will reset the slot source-of-current-value to the supplier of the new value, and the slot items-this-value-depends-on to NIL. Finally the interpreter will set the values of the items listed in items-depending-on-this-value to undefined.

Frames also make it possible to specify reflective computation about a data-item. They allow to define subroutines which are activated by specific events in the computation. Typically, these implement tasks such as consistency maintenance, documentation and explanation facilities, acquisition of new data, appropriate communication of data, garbage collection, etc. Figure 4 shows an example.

```
ITEM a name
      .
      . }        reflective data (cf. above)
      .
   before-retrieved: a procedure to be executed before the item is
                      asked for its value
   after-retrieved: a procedure to be executed after the item is asked
                     for its value
   when-initialised: a procedure to be executed when the item is
                      initialised
   before-modified: a procedure to be executed before the value of the
                     item is modified
   after-modified: a procedure to be executed after the value of the
                    item is modified
   when-displayed: a procedure to be executed when the item is displayed

ITEM salary of John
   value: 23
      .
      . }      reflective data (cf. above)
      .
   before-retrieved: a procedure which checks in the slot "showable"
                      whether the value of this slot is public
   after-retrieved: a procedure which resets the slot
                     when-last-accessed to the present date
   when-initialised: a procedure which sets up procedures to pay John's
                      salary every month
   before-modified: a procedure which checks whether: (i) the requestor
                     is allowed to modify the salary of John, (ii)
                     whether the proposed salary is reasonable, etc.
   after-modified: a procedure which handles the consistency of the
                    knowledge base
   when-displayed: a procedure which prints the salary of John with
                    a $ sign in front of the amount
```

Fig. 4. A frame includes reflective procedures about a data-item.

Several reflective procedures are associated with the knowledge item representing John's salary. When the salary of John is requested, the system first checks whether the salary of John is a showable item. If not, the value of John's salary is not returned. If positive, the present value of the salary of John is returned and the system records the fact that somebody accessed this item at the current moment. When the salary of John is given a value for the first time, the system sets up the procedures that take care that John's salary is paid every month.

When somebody wants to modify the value for the salary of John the system checks whether (i) the requestor is allowed to modify the salary of John and (ii) whether the proposed salary is acceptable. After the change is made, the system takes the appropriate actions to ensure the consistency maintenance of the knowledge base. All items in the knowledge base whose value depends upon the value of John's salary will have to be rejected. For example, John's social security contribution will receive a new value.

The success of the reflective facilities provided by frames is unquestionable. The idea has been incorporated in almost all commercially available expert system shells (cf. KEE (Intellicorp,1985) or ART (Clayton,1985)).

4. Reflective Computation about the Program of the Executer: Dynamically Modifiable Interpreters

A major advantage of a language with meta facilities is that it is **open-ended**, i.e. that it can be adapted to user-specific needs. For example, before meta-interpreters became popular in the field of logic programming, variants of the language were obtained by introducing extensions to the language. Such an extension involved the definition of a new syntactical construct and the adaptation or rewriting of the interpreter code. Extensions were proposed for handling defaults, for handling multiple contexts, for uncertain reasoning, etc.

Meta-interpreters make it possible to define extensions to the language in a more flexible way. They are designed to facilitate static manipulations of the interpreter (cf. chapter II). A meta-interpreter for PROLOG, for example, makes it possible to define a special interpreter for a program by modifying an interpreter for PROLOG written in PROLOG. It is thus possible to provide only a very simple and pure deduction mechanism in the (kernel-) language and to obtain more complex behavior (non-standard logics) if needed, by exploiting the meta facilities.

The specific contribution of reflection to making languages open-ended is to allow an explicit investigation of these variants. Reflective computation makes it possible to switch between variants of the interpreter from within the computation itself. It makes it possible to dynamically create variants of the interpreter and immediately use them.

For example, in a logic-based language with reflective facilities, such as F.O.L. (Weyhrauch,1980), each program (or theory) can have a reflective counterpart (or meta-theory). This meta-theory embodies a logical interpretation that can be given to the theory. It incorporates an explicit causally connected representation of the language in the language itself (the theory META) and the program-specific modifications that are made to this default-interpreter. Because of this reflective facility greater expressive and deductive power is obtained while retaining (globally) the standard semantics of logic.

The reflective facilities of F.O.L. for example overcome the problem of deduction from a single fixed theory. In one application several meta-theories, each implementing a variant of the default-interpreter (the theory META) can coexist. One meta-theory may implement a default-behavior, another meta-theory may implement reasoning with uncertainties, etc. An application is able to investigate the provability of a theorem in these different meta-theories. Figure 5 illustrates two such meta-theories (example inspired by Bowen and Kowalski,1982).

```
META: provable(T,F) :- theorem(T,F).
      provable(T,and(F,G)) :- provable(T,F), provable(T,G).
      provable(T,F) :- clause(T,F,G), provable(T,G).

VARIANT-1: provable(T,F) :- theorem(T,F).
           provable(T,and(F,G)) :- provable(T,F), provable(T,G).
           provable(T,F) :- clause(T,F,G), provable(T,G).
           theorem(T,lives-in(X,Y)) :- provable(T,works-in(X,Y)),
                                       not(provable(T,lives-in(Z,Y))).

VARIANT-2: provable(T,F) :- theorem(T,F).
           provable(T,and(F,G)) :- provable(T,F), provable(T,G).
           provable(T,F) :- clause(T,F,G), provable(T,G).
           theorem(T,lives-in(X,Y)) :-
                                 provable(T,lives-in(X,spouse(Y))),
                                 not(provable(T,lives-in(Z,Y))).
```

Fig. 5. Two variants on the default meta-interpreter META. Variant 1 and variant 2 both implement a form of non-monotonic reasoning.

Variant-1 and variant-2 both implement a specific example of default-reasoning. In variant-1 it is possible to deduce the fact that a person Y lives in a city X, if Y works in X, and we cannot prove that Y lives in a city Z. In variant-2 it is possible to deduce the fact that a person Y lives in a city X, if the spouse of Y lives in X and we cannot deduce that Y lives in a city Z. By means of reflection it is possible to refer to these alternative interpreters during computation. The technical realisation of this is discussed in the next chapter. For example, the clause

```
lives-in(Z,X):- not(reflect(META,lives-in(W,X))),
                reflect(VARIANT-1,lives-in(Y,X))),
                reflect(VARIANT-2,lives-in(Z,X))).
```

states that if it is not provable in the default-meta-theory that it is a theorem that person X lives in a city W, and in variant-1 it can be deduced that it is a theorem that X lives in a city Y, and in variant-2 it can be deduced that it is a theorem that X lives in city Z, then conclude that X lives in city Z (the city where the spouse lives).

This example demonstrates that reflection makes it possible to dynamically switch between alternative designs of the language interpreter. This dynamic open-endedness makes a language with reflective facilities an ideal tool for language design. A minimal kernel of the language, incorporating reflective facilities, can be implemented. Variants and extensions of the language can be constructed and studied from within the language itself by means of reflection.

5. Reflective Computation about the Data of the Executer: Handling Exceptions

In the deterministic model of computation, the sequence of steps that have to be taken during computation is fixed by the program of a computational system. The type of problems that are solved in deterministic languages, are problems for which it is possible to specify an algorithm that leads the computational system to the solution. However, in real-world systems, there are sometimes exceptional situations were the state of things is not exactly as expected by the algorithm. Examples of such exceptional situations are inconsistent data, incomplete data, errors, loops, and deadlocks.

Some programming languages have responded to the need for handling these exceptions by supporting reflective constructs such as catch & throw, escape and exit. We discuss here a concrete use of such a reflective construct provided by Zeta-LISP. This facility is refered to as "condition signalling and handling". It makes it possible to set up a monitor which temporarily watches the computation and checks whether a certain event happens. This monitor may for example be on the look out for whether an error occurs, whether a certain variable is set, or whether a variable obtains a specific value. If the event takes place, the monitor will activate a procedure, called a "handler", which can alter the flow of computation.

Consider a function "search-graph" defined as

```
(defun search-graph (node attribute)
      (if (has-attribute node attribute)
          node
          (if (father-node node)
              (search-graph (father-node node) attribute)
              NIL)))
```

The purpose of the function is to search an inheritance-graph of nodes, in order to find the father-node of a given node which has an attribute with a given name. When the user by accident defined an inheritance-graph with a circularity in it, this will cause the function search-graph to loop. Figure 6 illustrates how this special event can be guarded by a condition signaller and handler.

```
(defun search-graph (node attribute)
   (catch 'circular
      (condition-bind ((sys:pdl-overfow
                           '(lambda (error-flavor-instance)
                                (circularity-checker
                                    ',node
                                    error-flavor-instance))))
         (if (has-attribute node attribute)
             node
             (if (father-node node)
                 (search-graph (father-node node) attribute)
                 NIL)))))

(defun circularity-checker (node error-flavor-instance)
       (do ((already-encountered ()
                                  (cons current-node already-encountered))
            (current-node node (father-node current-node)))
           ((or (member current-node already-encountered)
                (null (father-node current-node)))
            (if (father-node current-node)
                (throw 'circular
                       (format t "The computation was halted due to an
                                  error in the inheritance-graph. Starting
                                  from node ~A, a circular path of
                                  father-nodes exists. Correct this
                                  immediately to avoid further problems."
                               current-node))
                (send error-flavor-instance :proceed :grow-pdl)))))
```

Fig. 6. Conditions in Zeta-LISP make it possible to alter the flow of control from within the computation.

Sys:pdl-overflow is one of the standard error events recognised by the Zeta-LISP interpreter. It is signalled when there is a stack-overflow. The condition-bind construct defines a local handler for this error. If the stack overflows in the interpretation of the body of the function search-graph, the form

```
(circularity-checker node error-flavor-instance)
```

will be executed. The function circularity-checker tests whether the graph is really circular (it might just be a very large graph). It puts the nodes that it encounters in the list "already-encountered". Note that this function was written in an iterative way, because it would otherwise also cause the stack to overflow. If the graph is circular, i.e. if current-node is a member of the list of already-encountered nodes, a message is printed and the interpretation of search-graph is halted. If the graph is not circular, i.e. if the current-node has no father-node, the computation will proceed from the error. The computation that was interrupted is continued with a larger stack.

Condition signalling and handling is used to implement limits on the available resources for computation, to implement active values, to implement default computation, to add "side-computation" to computation (e.g. stepping, tracing), etc.

So, reflection gives systems an escape mechanism for handling the exceptional situations that can occur when operating in a real environment. Even more, reflective computation provides the technical means for constructing self-understanding and self-debugging programs. Reflective computational systems can analyse their own computation. This makes it technicallsy possible for systems to recognise error-situations, such as loops and deadlocks. Reflective computational systems also modify their own computation. This makes it technically possible for systems to recover from error situations.[4] A system may, when it has recognised an error, change the environment and the continuation of its own object-computation such that this computation may proceed from the error.

6. What Not To Do with Reflection: Dangers of Reflection

The previous sections showed that reflection provides interesting solutions to many programming problems. Although the examples unquestionably prove that reflective computation can be very useful, there is certainly a danger attached to its use.

There are limits to controlling and understanding reflective computational systems. A reflective system is able to make modifications to itself. This means that some of the control over computation is actually shifted from the programmer to the system. Consequently programming the computation of such a system is less straightforward. It involves programming the object-level computation, reflective computation, and so forth. This may become an extremely difficult task, since these different levels of computation actually act upon one another. The computation of a specific level of computation is not only determined by the code for that level, but also by the code of the reflective levels above that level.

The shift of control also implies a loss of information. For example, if we build a system that is able to handle exceptional or erroneous situations, the programmer/user is no longer informed about these situations. It might be that the fact that for example a symbol is not bound, indicates to the programmer that his program is not correct. Often a programmer wants to know when and why a variable is not as it should be. This certainly suggests not to use reflection for exception-handling during the development phase of a system.

Another problem in reflective systems is to understand what a computational system actually does. Reflection makes the semantics of a system more explicit to the system itself. But at the same time, since a system may modify itself, this semantics becomes less clear to the programmers and users of a system. Actually, the semantics of a language becomes opened by reflection. The (object-) program of a system is no longer a means for understanding the behavior of a system. It might for example be that at run-time a different program is executed, or that a different interpreter is executing the program. So in a reflective language the semantics of sentences in the language is pulled down.

Clearly reflection might sometimes be a dangerous facility. However, it would be wrong to therefore conclude that the programming community should discard reflection. All powerful engineering tools can be dangerous (cf. for example the LISP-machine). One solution to the problem is a discipline in the use of reflection. What this discipline should look like is not yet known. Only now that reflection becomes better understood, can this become a topic for further research. We need to experiment with reflection in order to distinguish its positive and negative uses. On the basis of such a study, safer (weaker) versions of reflective facilities will have to be designed.

7. Conclusions

This chapter presented an informal discussion of the uses of reflection. Four concrete programming problems requiring reflective computation were discussed:

- reasoning about control,
- handling exceptional situations occuring during computation,
- making variants of a language and its interpreter from within the language,
- giving systems data about the use, structure, and validity of their information,

As the examples demonstrate, having access to representations of aspects of the system itself during computation can be very useful. It is therefore that we believe that reflective computation should be supported by programming languages. Languages with a reflective architecture are designed exactly for this purpose.

Chapter III present some examples of languages with a reflective architecture which have already been built.

NOTES

[1] In languages adopting the non-deterministic model of computation, **theoretically** different sequences of action can be followed. However, **practically** the interpreter of such a language imposes a particular flow of control such that the same sequence of actions is adopted for the same program/input pair. However, different flows of control remain potentially present. In PROLOG, for example, these different sequences may all be explored through the backtracking mechanism.

[2] We call TEIRESIAS' rules reflective rules, instead of meta-rules, as Davis does, because this conforms with the terminology used throughout this dissertation. TEIRESIAS indeed has a reflective architecture, as opposed to other rule-based systems, which only incorporate meta-knowledge. Albeit that the reflective architecture of TEIRESIAS, as described in (Davis,1982), could have been designed with more powerful capabilities. TEIRESIAS' reflective rules are only used to decide which rules to invoke next during computation. Reflective rules cannot modify or create new object-level rules.

[3] The main obstacle for the construction of self-debugging systems is the issue of side-effects. If programs would be side-effect free, a system could (in principle) when encountering an error, inspect the current state of computation, identify the errant code, modify and recompile it and resume the computation at the point of the stack where this procedure was used.

[4] Logicians have also since long been interested in self-referencing data. Research in autoepistemic logics studies self-reference from a theoretical point of view. The question this research tries to answer is what the logically "correct" foundations for self-referencing data are. Several axioms sets for studying knowledge bases containing this type of data have been proposed. They are similar to the set S5 shown below.

A1. All tautologies of the propositional calculus
A2. [Kp and K(p => q)] => Kq
A3. Kp => p
A4. Kp => KKp
A5. not(Kp) => K(not(Kp))

"=>" stands here for logical implication, while p and q represent facts. The K-operator can be read as "knows" or "probable" or "belief" (Perlis,1987). However, notice that most of these theories are proposed as theories for studying the knowledge content of a system. They represent the reasoning about the knowledge contents of a system by a second system (e.g. the theorist). In order to speak of reflection these theories should be attributed to the systems themselves.

CHAPTER III

Examples of Reflective Architectures

1. Introduction

The previous chapter argued that a lot of programming problems might profit from reflective facilities. Some programming languages have responded to this need by incorporating reflective constructs without recognising reflection as a primary programming concept. The reflective constructs they support are not part of the kernel design of the language. Consequently, these languages only support a finite set of reflective constructs, designed and implemented in an ad hoc way. Often different dialects of the language support different reflective functionalities.

A common theory of the representation, the use and the role of reflective computation is clearly not yet established. This is precisely the role and contribution of reflective architectures. Reflective architectures seek to identify general principles of reflection and to support them. They recognise the importance of reflection for computational systems and present a general framework for the representation and use of reflective computation. They do not restrict the possibilities for reflection to a fixed number of constructs, but provide mechanisms for constructing reflective functions the way one constructs regular functions. Consequently, they provide a more modular and open-ended solution.

Actually reflective architectures offer a completely new paradigm for thinking about computation 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 about the external domain, while the task of the reflective level is to solve problems about the object computation. A computational system implemented in a reflective architecture has data representing aspects of itself. These data can be accessed and manipulated like ordinary data. Because of the causal connection link between these data and the things they represent, such manipulations are reflected in the status of the actual system.

For example, in a reflective architecture one can temporarily associate reflective computation with a program such that during its interpretation tracing is performed. Assume a session with a rule-based system has to be traced. The goal is to receive a trace of the rules that are applied. This can be achieved in a language with a reflective architecture by stating a reflective rule such as[1]

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

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.

Reflective architectures introduce fascinating new perspectives on programming. The computation of real world systems is flooded with pieces of reflective computation. Examples are keeping statistics, documentation, type-declaration, interfacing (e.g. graphical output, mouse input, etc), debugging, tracing, breaking, stepping, active values, consistency maintenance, etc. Reflective architectures provide a means to implement these activities in a more modular way. It makes the implementation effort easier, and produces more elegant and more adaptable computational systems.

We are now ready to move to a more technical level. The following sections illustrate reflective architectures in procedural languages, logic-based languages and production rule systems. The discussions are based on a number of existing languages with a reflective architecture, namely, 3-LISP, F.O.L, METAPROLOG, SOAR and TEIRESIAS. The purpose of the discussions is not to give a complete and accurate description of these languages, but to present concrete illustrations. Consequently, the accounts of these languages may sometimes be oversimplified or imprecise. The discussions do remain faithful to the philosophy and overall design of the languages. This remark holds also for the other chapters in which these languages serve as illustrative vehicle.

2. Reflection in a Procedural Language

3-LISP is a dialect of LISP designed by B. C. Smith (1982). 3-LISP has a reflective architecture. It has two sorts of functions. **Simple functions**, constructed with define-SIMPLE or lambda-SIMPLE, specify regular object-level computation. **Reflect functions**, constructed with define-REFLECT or lambda-REFLECT, specify reflective computation. Reflective functions bring the computation temporarily to the level at which the interpreter of the current computation is run. Reflective functions take a number of quoted arguments. They represent, so to say, a local special interpreter for their arguments. Reflective functions have a representation of the current computation: they have access to two extra variables, called "env" and "cont", which by default represent the environment (a list of bindings) and the continuation at the time the reflect function is called.

A reflect function is able to inspect these (e.g. checking variable bindings) and to modify these (e.g. changing the continuation or changing variable bindings). The env and cont variables are causally connected to the real environment and continuation of the system, such that the results of this reflective computation are reflected in the system's future object-level computation.

Figure 7 shows a very simple reflective 3-LISP program. Note that the purpose of this example is to introduce the design of the 3-LISP language. It is thus not written to serve as an example of a programming problem for which a reflective architecture should be used (cf. chapter III and VII). Note also that to convey the essence of the notion of a reflective architecture, we took the liberty to simplify aspects of 3-LISP as compared to (Smith,1982)[2]. The code represented below is conform with the Common-LISP conventions.

```
(define-REFLECT boundp-else-bind-to-one (symbol &optional env cont)
      (let ((value (binding symbol env)))
         (funcall cont
                  (if value
                      value
                      (rebind symbol 1 env)))))
```

Fig. 7. A reflective procedural program.

When the above function is called, for example in

```
(let ((x 36))                                         (1)
   (/ x (boundp-else-bind-to-one y)))
```

The evaluation returns 36 after reflection (because the symbol y is not bound in this environment). On the other hand, the evaluation of

```
(let ((x 36)
      (y 12))
   (/ x (boundp-else-bind-to-one y)))
```

returns 3. These results can be explained with the help of some of the successive states of the interpretation process.

3-LISP has a continuation-passing interpreter. One state of the interpreter is characterised by

(i) A level of computation.
Level 0 is the level at which ordinary procedures about the domain are run. Level 1 is the level at which the interpreter and the reflective procedures of level 0 are run. Level 2 runs the reflective procedures of level 1, i.e. the procedures that reflect upon the reflective activities of level 0, and so on.

(ii) The expression that is being evaluated by the interpreter at that moment.

(iii) The list of bindings (called the environment) existing at that moment.

(iv) The continuation of this evaluation.
The continuation is the lambda-function that will be applied to the result of the current evaluation. If the continuation and the expression

that is being evaluated are both NIL, the final result is returned.

When evaluating expression (1) the state of the interpreter is initially as follows:

```
LEVEL: 0
EXPRESSION: (let ((x 36))
              (/ x (boundp-else-bind-to-one y)))
ENVIRONMENT: ()
CONTINUATION: NIL
```

First the bindings specified in the let-construct are made:

```
LEVEL: 0
EXPRESSION: (/ x (boundp-else-bind-to-one y))
ENVIRONMENT: ((x.36))
CONTINUATION: NIL
```

The subexpression

```
(boundp-else-bind-to-one y)
```

is evaluated, the continuation being the division by the result of the value of x:

```
LEVEL: 0
EXPRESSION: (boundp-else-bind-to-one y)
ENVIRONMENT: ((x.36))
CONTINUATION: (lambda-SIMPLE (c) (/ x c))
```

Note that reflective functions do not evaluate their arguments. A reflective function such as boundp-else-bind-to-one represents a local deviating interpreter. It explicitly prescribes the interpretation of a particular piece of code (here a symbol). Calling a reflective function causes a jump to the reflective level.

```
LEVEL: 1
EXPRESSION: (let ((value (binding symbol env)))
              (funcall cont
                       (if value
                           value
                           (rebind symbol 1 env))))
ENVIRONMENT: ((symbol.y)
             (env.((x.36)))
             (cont.(lambda-SIMPLE (c) (/ x c))))
CONTINUATION: NIL
```

"Symbol" is in the function-call of boundp-else-bind-to-one bound to the symbol y. A reflective function also binds two extra arguments, named "env" and "cont" at the moment of its call. These are optional arguments, which have a default-value. In case values for "env" and "cont" are absent in the call of a reflective function, they will respectively be bound to the current environment (list of bindings) and current continuation of the interpretation.

First, the bindings of the let-construct are made. "Binding" is a function that returns the binding of a symbol in an environment. If the symbol is unbound it returns NIL. Since the symbol y is not bound in "env", representing the environment of the object-computation, the symbol value will be bound to NIL in the environment of the reflective computation:

```
LEVEL: 1
EXPRESSION: (funcall cont
                     (if value
                         value
                         (rebind symbol 1 env)))
ENVIRONMENT: ((value.NIL)
              (symbol.y)
              (env.((x.36)))
              (cont.(lambda-SIMPLE (c) (/ x c))))
CONTINUATION: NIL
```

Funcall sets up a new evaluation-process for the object-level by means of the current continuation "cont" and the if-expression. The if-expression is first evaluated, the result being passed to

```
(lambda-SIMPLE (c)
        (funcall cont c))
```

We get:

```
LEVEL: 1
EXPRESSION: (if value
                value
                (rebind symbol 1 env))
ENVIRONMENT: ((value.NIL)
              (symbol.y)
              (env.((x.36)))
              (cont.(lambda-SIMPLE (c) (/ x c))))
CONTINUATION: (lambda-SIMPLE (c)
                  (funcall cont c))
```

Because the symbol value is NIL, we get:

```
LEVEL: 1
EXPRESSION: (rebind symbol 1 env)
ENVIRONMENT: ((value.NIL)
              (symbol.y)
              (env.((x.36)))
              (cont.(lambda-SIMPLE (c) (/ x c))))
CONTINUATION: (lambda-SIMPLE (c)
                 (funcall cont c))
```

The function "rebind" is defined such that

```
(rebind symbol value environment)
```

modifies the environment "environment" to contain a binding of the symbol "symbol" to the evaluation of "value". It returns the new value. The symbol y is bound to 1 in the environment "env" (in italics):

```
LEVEL: 1
EXPRESSION: (funcall cont 1)
ENVIRONMENT: ((value.NIL)
              (symbol.y)
              (env.((x.36)(y.1)))
              (cont.(lambda-SIMPLE (c) (/ x c))))
CONTINUATION: NIL
```

Finally the continuation "cont" of the object-level, which is the lambda-function

```
(lambda-SIMPLE (c) (/ x c))
```

is applied to the number 1. Calling the continuation of the lower level makes the interpretation descend one level, thereby reflecting the values of the env and cont variables in the actual environment and continuation of the computation of the level below. Consequently, the procedure divide continues its computation with the modified runtime environment, in which y is bound to 1:

```
LEVEL: 0
EXPRESSION: (/ x 1)
ENVIRONMENT: ((x.36)(y.1))
CONTINUATION: NIL
```

The evaluation of this last expression returns 36.

This example illustrates the architecture for reflection of 3-LISP: any 3-LISP program can specify reflective computation by means of a reflective function application. The evaluation of this reflective sub-expression brings the interpretation of the program one level up, being the level where the interpretation of the program was run until that moment. This level reasons about and acts upon the environment and continuation of the level below.

The 3-LISP interpreter takes care of the causal connection between the system and the representation it has of itself. Whenever the program specifies reflective computation (by calling a "reflect" function), the interpreter constructs variables denoting the environment and the continuation at that moment of interpretation. The reflective computation can manipulate these variables. When the reflective computation reactivates the object-computation, the interpreter continues the computation at the level below, after reflecting the bindings of these variables in the actual environment and continuation.

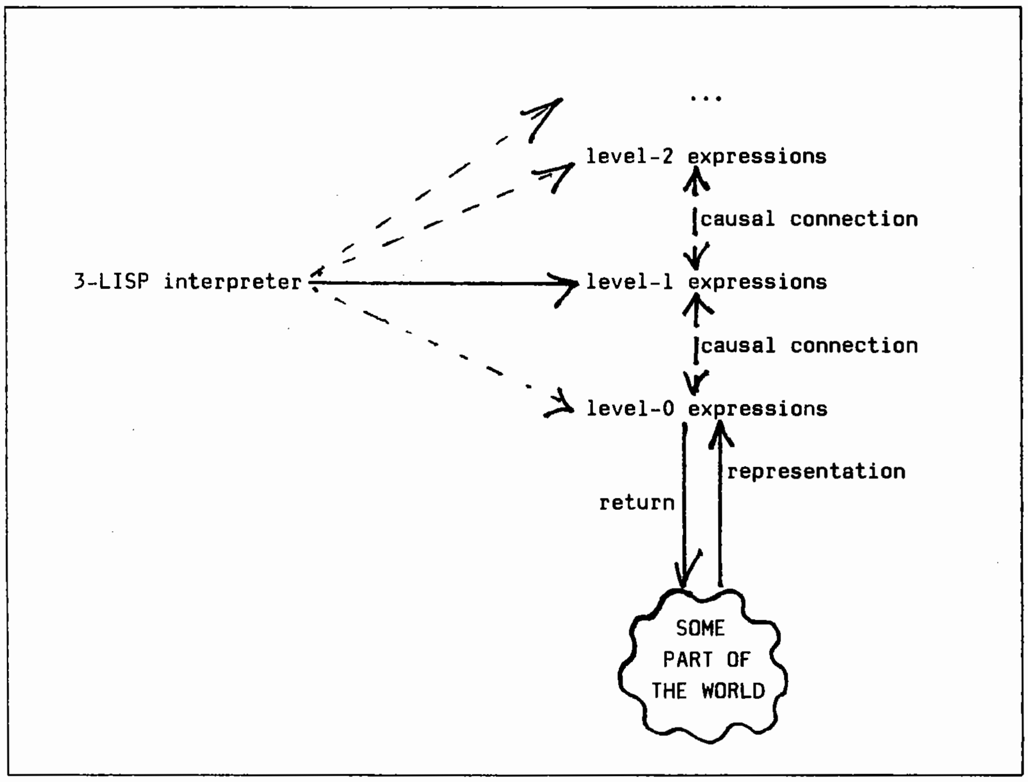


Fig. 8. The interpretation of a 3-LISP program.

The actual 3-LISP interpreter constantly moves from one level of activity to another as prescribed by the reflective expressions. When moving from one level to another, it takes care of the causal connection link between these levels: (i) when going up, by representing the state of the lower level at the higher level in the variables cont and env, (ii) when going down, by reflecting these variables in the state of computation at the lower level again.

3. Reflection in a Logic-Based Language

This section discusses an example of a reflective architecture in a logic-based language. The example is based on the languages **F.O.L.** (Weyhrauch,1980) and **META-PROLOG** (Bowen,1985). Logic-based languages express programs by means of theories. The theory called my-theory in figure 9 contains some facts and inference rules that I believe in. The theory called john's-theory contains facts and inference rules John believes in. Meta-t is a meta-theory for john's-theory (or any other theory which models the beliefs of somebody other than myself).

The data (or variables) of meta-t are representations of the theorems and clauses of another theory T. This means that the predicates of meta-t range over predicates and clauses of T. The example adopts the syntactical conventions explained in section 2.5 of chapter I. Note particularly the final rule of meta-t which states that a clause from my theory may be used to prove a theorem F in a theory T. If this happens,

```
        theorem(T,F)
```

is asserted in meta-t.

```
my-theory: mortal(X) :- human(X).

john's-theory: human(X) :- greek(X).
               greek(socrates).
               P :- reflect(meta-t,P).

meta-t: provable(T,F) :- theorem(T,F).
        provable(T,and(F,G)) :- provable(T,F), provable(T,G).
        provable(T,F) :- clause(T,F,G), provable(T,G).
        provable(T,F) :- clause(my-theory,F,G), provable(T,G),
                         assert(theorem(T,F)).
```

Fig. 9. A reflective logic program.

This implements the autoepistemic rule that I may assume that somebody else uses the same inference rules (but not the same facts) as I do, complemented by inference rules specific to him. An attempt to prove in john's-theory

```
mortal(socrates)
```

results, when the other clauses have failed to prove the goal, in the exploration of the last clause of john's-theory. This means that the interpreter proceeds with an attempt to prove

```
reflect(meta-t,mortal(socrates))
```

Reflect is a special predicate that attempts to prove a theorem in a meta-theory. If the meta-theory succeeds in proving the theorem, this result is reflected in the theory. For the above example, it means that the fact

```
mortal(socrates)
```

will be asserted in john's-theory, if the theory meta-t succeeded in proving the theorem

```
provable(john's-theory,mortal(socrates))
```

We again try to explain this result by means of some of the successive states of the interpreter. One state of the interpreter is characterised by the following information:

(i) A level of computation (here called deduction).
Level 0 represents deduction about some external domain. Level 1 represents reflective deduction about level 0, etc.

(ii) The theory in which deduction happens.
This theory consists of the data-base of already known facts and clauses.

(iii) The current goal,
The current goal is the goal the interpreter is currently trying to solve. A goal is solved if it is a theorem in the theory. If it is not a theorem, the interpreter (a) retrieves the set of clauses in the theory whose left-hand matches this goal, (b) adds the right-hand of one of those clauses to the list of goals (see iv) (c) adds a frame to the stack (see vi) for each of the other clauses of the set.

(iv) The list of goals that remain to be proved in order to prove the overall goal.
If this list is empty, the interpreter succeeded in proving the overal goal of this level. If at that moment the current level is level 0, the interpreter halts computation, else, the computation at the level below is continued.

(v) The bindings that hold at this moment in the proof.

(vi) The stack,
The stack holds the backtracking information for the proof. It consists of frames which specify a theory, a list of goals and a list of bindings. If the interpreter fails to prove the current goal, it pops a frame from this stack and tries to bring that interpretation state to a good end.

The initial state of interpretation is as follows:

```
LEVEL: 0
THEORY: human(X) :- greek(X).
        greek(socrates).
        P :- reflect(meta-t,P).
GOAL: mortal(socrates).
GOALS: ()
BINDINGS: ()
STACK: ()
```

All clauses that can possibly solve this goal are explored sequentially. Since there is not enough information in john's-theory to prove that socrates is mortal, the deduction proceeds in an attempt to prove

```
reflect(meta-t,mortal(socrates))
```

So the interpreter continues with:

```
LEVEL: 0
THEORY: human(X) :- greek(X).
        greek(socrates).
        P :- reflect(meta-t,P).
GOAL: reflect(meta-t,P)
GOALS: ()
BINDINGS: ((P.mortal(socrates)))
STACK: ()
```

reflect(T,F) is a special predicate that attempts to prove provable(ct,F), where ct is the current theory, in the theory T. This means that the goal

```
provable(john's-theory,mortal(socrates))
```

is stated in meta-t:

```
LEVEL: 1
THEORY: clause(my-theory,mortal(X),human(X)).
        clause(john's-theory,human(X),greek(X)).
        theorem(john's-theory,greek(socrates)).
        clause(john's-theory,P,reflect(meta-t,P)).
        provable(T,F) :- theorem(T,F).                          (1)
        provable(T,and(F,G)) :- provable(T,F), provable(T,G).   (2)
        provable(T,F) :- clause(T,G,F), provable(T,G).          (3)
        provable(T,F) :- clause(my-theory,G,F), provable(T,G),  (4)
                         assert(theorem(T,F)).
GOAL: provable(john's-theory,mortal(socrates)).
GOALS: ()
BINDINGS: ()
STACK: ()
```

Meta-t has a representation of john's-theory and my-theory. It knows how a theory T has to be interpreted ((1) .. (4)), and knows what theorems have already been proved. Meta-t is able to modify john's-theory (as well as my-theory) and its deduction process by means of these causally connected representations. The deduction will arrive at (4) where the first subgoal

```
clause(my-theory,F,G)
```

unifies with

```
clause(my-theory,mortal(X),human(X))
```

leading to:

```
LEVEL: 1
THEORY: clause(my-theory,mortal(X),human(X)).
        clause(john's-theory,human(X),greek(X)).
        theorem(john's-theory,greek(socrates)).
        clause(john's-theory,P,reflect(meta-t,P)).
        provable(T,F) :- theorem(T,F).
        provable(T,and(F,G)) :- provable(T,F), provable(T,G).
        provable(T,F) :- clause(T,F,G), provable(T,G).
        provable(T,F) :- clause(my-theory,F,G), provable(T,G),
                         assert(theorem(T,F)).
GOAL: provable(T,G)
GOALS: ()
BINDINGS: ((T.john's-theory)(F.mortal(socrates))
           (G.human(X))(X.socrates))
STACK: ()
```

This goal will be solved using facts and inference rules of john's-theory, leading to the following state of the interpreter (see italics):

```
LEVEL: 1
THEORY: clause(my-theory,mortal(X),human(X)).
        clause(john's-theory,human(X),greek(X)).
        theorem(john's-theory,greek(socrates)).
        clause(john's-theory,P,reflect(meta-t,P)).
        provable(T,F) :- theorem(T,F).
        provable(T,and(F,G)) :- provable(T,F), provable(T,G).
        provable(T,F) :- clause(T,F,G), provable(T,G).
        provable(T,F) :- clause(my-theory,F,G), provable(T,G),
                         assert(theorem(T,F)).              (*)
        theorem(john's-theory,mortal(socrates)).            (1)
GOAL: /
GOALS: ()
BINDINGS: ((X.socrates))
STACK: ()
```

Note that (1) is asserted because of the assert-predicate in clause (*). Note that an assert predicate is always true. Reflection principles such as those specified in figure 10 are responsible for the communication of results between the meta-theory and the theory.

```
For all W wff:

                    (In meta-t)          theorem(W)
                                        ____________

                    (In t)                    W

For all F,G wff:

                    (In meta-t)          clause(F,G)
                                        ____________

                    (In t)                 F :- G
```

Fig. 10. Two reflection principles of F.O.L.

This means that john's-theory is affected by the reflective deduction of meta-t. It will contain the new fact

```
mortal(socrates).
```

Finally

```
P :- reflect(meta-t,P).
```

succeeds, such that the initial goal is proven in john's-theory, and in addition there has been a side-effect. Note that it would not have been possibleto deduce

```
mortal(socrates).
```

in my-theory.

This example illustrates an architecture for reflection in a logic-based language. Programs are represented by means of theories. A theory may have one or more meta-theories which specify deduction about the theory. The language interpreter has mechanisms that are responsible for the causal connection between a theory and a meta-theory. These mechanisms guarantee the communication of results between the two levels, i.e they specify how to reflect the results of meta-theory computation in object-theory computation (and vice versa) and are therefore called reflection principles

(Weyhrauch,1980).

Just like in 3-LISP, the interpreter can switch from one level to another as prescribed by the program. Since all information of the object-level is also present at the reflective level (because of the reflection principles), a proof in the object-level could be completely simulated at the reflective level. Nevertheless, object-level computation remains useful because it is more efficient (it is "compiled" in some sense).

4. Reflection in a Production Rule System

We now illustrate a reflective architecture in a rule-based system. The example is inspired by systems like **SOAR** (Laird, Rosenbloom and Newell,1984) and **TEIRESIAS** (Davis,1982). Computational systems are implemented in a rule-based language by means of production rules. A production rule has a condition and an action part. Computation consists of "firing" these rules on a working-memory of data. A rule can be fired if its condition part matches the working-memory. When a rule is fired, the actions specified in the action part are executed. In the language presented here, the computation halts when the working-memory matches a certain goal.

The executing process of a rule-based system is called the inference engine. The task of the inference engine is to determine which rules are relevant to a given working-memory configuration and to fire these. In some cases deciding upon the rules to be fired requires computation. These control problems are handled in standard rule-based systems such as EMYCIN and OPS5 by a wired-in control structure coded into the LISP implementation of the inference engine.

Rule-based systems with a reflective architecture, such as TEIRESIAS, handle them by means of reflective computation. When an "impasse" in the inference process occurs, a reflective production-rule program is generated that tries to resolve the impasse[3]. A typical example of an impasse is when there is more than one rule which matches with data in the current working-memory.

The rules of this reflective program are called reflective rules. Reflective rules are rules which are about the inference process itself (e.g. about impasses). The working-memory these rules operate on represents the computation of the object-level rule-program. There are domain-independent and domain-dependent reflective rules. An example of a domain-independent reflective rule from TEIRESIAS is

```
IF 1) there are rules which do mention
      the current goal in their premise,
   2) there are rules which do not mention
      the current goal in their premise,

THEN it is definite that the former should be done before the latter.
```

An example of a domain-dependent reflective rule from TEIRESIAS is

```
IF 1) the infection is a pelvic-abscess, and
   2) there are rules which mention in their
      premise enterobacteriaceae, and
   3) there are rules which mention in their premise grampos-rods,

THEN there is suggestive evidence (.4) that the former should be
     done before the latter.
```

Domain-dependent reflective rules have priority over domain-independent rules.

When the domain-dependent and independent reflective rules have succeeded in solving the impasse, the reflective computation will have affected the object computation. Consider for example the reflective system in figure 11.

```
WORKING-MEMORY: ((o1.True)(o2.True)(o3.True)(p1.)(p3.)(p2.)...)

GOAL: ((p1.True) (p3.True))

RULE MEMORY:  (1) IF o1 and o2
                  THEN set(p1,True) and set(p2,False)
              (2) IF o3
                  THEN set(p2,True) and set(p1,False)
              ...

REFLECTIVE RULE MEMORY: (3) IF error-flag-1
                            THEN set(data-elm(o2),False) and
                                 set(rule-to-be-fired(2),True)
                        (4) IF satisfied(1) and
                               satisfied(2)
                            THEN set(error-flag-1,True)
```

Fig. 11. A reflective production rule program.

The computation is in an impasse, because both rule (1) and rule (2) match the current working-memory. The system will try to solve this impasse by initiating a reflective production rule program. The goal of this reflective program is

```
rule-to-be-fired(?rule) = True
```

The program in figure 11 incorporates domain-dependent reflective rules which might help to solve this impasse. For example the reflective rule

```
IF satisfied(1) and satisfied(2)
THEN set(error-flag-1,True)
```

says that, when both rule 1 and rule 2 can be fired, this is a special event in the object-level inference process (note that rule 1 and 2 propose contradictory actions). Consequently, the data-element error-flag-1 has to be set true. This will enable rule 3 to fire, which modifies the status of the object-level inference process (think of rule 3 as an error-handling procedure).

When the domain-dependent and independent reflective rules have succeeded in solving the reflective subgoal of the above example, the reflective computation will have affected the object computation by determining which of the two competing rules will be applied and by modifying the status of the working-memory.

We again take a closer look at the successive states of the interpreter in this example. The interpreter incorporates a stack of contexts, corresponding to the reflective levels that have been activated. The computation halts when this stack is empty. The interpreter always works on the top-level context. One context of the interpreter is characterised by:

(i) The level of rule firing. The lowest level represents the object-computation, level 1 reflective computation, level 2 reflective-reflective computation, and so on.

(ii) An unordered collection of production rules.

(iii) An unordered collection of domain-specific reflective production rules.

(iv) The goal of the computation.

(v) The current state of the working-memory.

(vi) The rules that match the current state of the working-memory.

(vii) The rule to be fired to generate the next state.

Computation consists of firing the rules of the context on the working memory of the context until the goal of the context is fulfilled. When a final state is found (one that achieves the goal), the upper context of the stack is popped, thereby communicating some results to the context below. Suppose the above program is being run and that the state of the interpreter at a certain point is as follows:

```
LEVEL: 0
RULE MEMORY:  (1) IF o1 and o2
                  THEN set(p1,True) and set(p2,False)
              (2) IF o3
                  THEN set(p2,True) and set(p1,False)
              ...
DOMAIN REFLECTIVE RULE MEMORY: (3) IF error-flag-1
                                   THEN set(data-elm(o2),False) and
                                        set(rule-to-be-fired(2),True)
                               (4) IF satisfied(1) and
                                      satisfied(2)
                                   THEN set(error-flag-1,True)
GOAL: ((p1.True)(p3.True))
WORKING-MEMORY: ((o1.True)(o2.True)(o3.True)(p1.)(p2.)(p3.)...)
SATISFIED RULES: (1) (2)
RULE-TO-BE-FIRED: ?
```

The above context represents an impasse because the inference engine can only fire one rule at a time and more that one rule is satisfied. The inference engine tries to solve this impasse by stating the goal

```
(rule-to-be-fired(?rule).True)
```

in a reflective context:

```
LEVEL: 1
RULE MEMORY:  (3) IF error-flag-1
                  THEN set(data-elm(o2),False) and
                       set(rule-to-be-fired(2),True)
              (4) IF satisfied(1) and
                     satisfied(2)
                  THEN set(error-flag-1,True)

              ... domain independent meta rules ... E.g.:

              (N) IF satisfied(?rule) and
                     mentions-in-premise(?rule,goal) and
                     satisfied(?another-rule) and
                     not(mentions-in-premise(?another-rule,goal))
                  THEN prefer(?rule,?another-rule)

DOMAIN REFLECTIVE RULE MEMORY: ... empty ...
GOAL: ((rule-to-be-fired(?rule).True))
WORKING-MEMORY: ((satisfied(1).True)(satisfied(2).True)
                 (goal((p1.True)(p3.True)).True)
                 (?rule.)(?another-rule.)
                 (rule(1).True)(rule(2).True)(error-flag-1.)...)
SATISFIED RULES: (4) ...
RULE-TO-BE-FIRED: (4)
```

Note that this reflective context represents in its working-memory aspects of the computation at the level below. In this context, rule 4 is fired, because it matches the data in the current state of the memory and has priority over domain-independent reflective rules such as rule (N). Rule 4 sets a memory element (in italics):

```
LEVEL: 1
RULE MEMORY:  (3) IF error-flag-1
                  THEN set(data-elm(o2),False) and
                       set(rule-to-be-fired(2),True)
              (4) IF satisfied(1) and
                     satisfied(2)
                  THEN set(error-flag-1,True)

              ... domain independent meta rules ... E.g.:

              (N) IF satisfied(?rule) and
                     mentions-in-premise(?rule,goal) and
                     satisfied(?another-rule) and
                     not(mentions-in-premise(?another-rule,goal))
                  THEN prefer(?rule,?another-rule)

DOMAIN REFLECTIVE RULE MEMORY: ... empty ...
GOAL: ((rule-to-be-fired(?rule).True))
WORKING-MEMORY: ((satisfied(1).True)(satisfied(2).True)
                (goal((p1.True)(p3.True)).True)
                (?rule.)(?another-rule.)
                (rule(1).True)(rule(2).True) (error-flag-1.True)...)
SATISFIED RULES: (3) ...
RULE-TO-BE-FIRED: (3)
```

This enables rule 3 to fire, which modifies the representation of the working-memory and rule-to-be-fired of the lower level:

```
LEVEL: 1
RULE MEMORY:  (3) IF error-flag-1
                  THEN set(data-elm(o2),False) and
                       set(rule-to-be-fired(2),True)
              (4) IF satisfied(1) and
                     satisfied(2)
                  THEN set(error-flag-1,True)

              ... domain independent meta rules ... E.g.:

              (N) IF satisfied(?rule) and
                     mentions-in-premise(?rule,goal) and
                     satisfied(?another-rule) and
                     not(mentions-in-premise(?another-rule,goal))
                  THEN prefer(?rule,?another-rule)

DOMAIN REFLECTIVE RULE MEMORY: ... empty ...
GOAL: ((rule-to-be-fired(?rule).True))
WORKING-MEMORY: ((satisfied(1).True)(satisfied(2).True)
                 (goal((p1.True)(p3.True)).True)
                 (?rule.rule(2))(?another-rule.)
                 (rule(1).True)(rule(2).True)(error-flag-1.True)
                 (rule-to-be-fired(2).True)(data-elm(o1).False)...)
SATISFIED RULES: ...
RULE-TO-BE-FIRED: ...
```

The reflective compution is halted, since the goal of this context is achieved (?rule is bound to rule(2)). The current context is popped from the stack, but its data are reflected in the context below (in italics):

```
LEVEL: 0
RULE MEMORY:  (1) IF o1 and o2
                  THEN set(p1,True) and set(p2,False)
              (2) IF o3
                  THEN set(p2,True) and set(p1,False)
DOMAIN REFLECTIVE RULE MEMORY: (3) IF error-flag-1
                                   THEN set(data-elm(o2),False) and
                                        set(rule-to-be-fired(2),True)
                               (4) IF satisfied(1) and
                                      satisfied(2)
                                   THEN set(error-flag-1,True)
GOAL: ((p1.True)(p3.True))
WORKING-MEMORY: ((o1.True)(o2.False)(o3.True)(p1.)(p3.)(p2.)...)
SATISFIED RULES: (1) (2)
RULE-TO-BE-FIRED: (2)
```

The object-level computation continues with the activation of rule (2).

This example illustrates the reflective architecture incorporated in systems like SOAR or TEIRESIAS. Programs in rule-based systems are sets of rules. Note that in our example reflection is controlled implicitly as opposed to being controlled explicitly as in the procedural and the logic example discussed before. Whenever an impasse occurs in the interpretation of a program, the interpreter sets up a reflective computation, which tries to solve this impasse. The interpreter also handles the causal connection between the object and reflective computation.

The moment the interpreter creates a context of computation, it represents the current state of computation of the level below in the working-memory of this new context. Consequently, domain independent and user-defined reflective rules manipulate this representation until they succeed in achieving the goal of the reflective computation. Once the goal is achieved, the reflective context is thrown away and the values of its working-memory elements are installed as the actual values of the objects of the context below.

5. The Power of a Reflective Language

An important advantage of languages with a reflective architecture over languages that provide reflective facilities, is that in the former the possibilities for reflection are open-ended. For example, although many LISP dialects provide reflective functions (e.g. catch and throw, condition signallers and handlers, call-current-continuation), there is an important difference between the reflective functionality provided by these languages and that provided by 3-LISP.

Programs in 3-LISP might define their own, local interpreter by means of a reflective function. The set of reflective functions that can be used is open-ended: reflective functions can be constructed just like object-level functions. They are constructed out of the same primitive functions of the language. However, they manipulate causally connected representations of the environment, stack and code of a function.

In a regular LISP, the run-time environment, continuation and code of the computation can also more or less be made accessible. Reflective constructs such as boundp, makunbound, catch and throw (cf. MacLISP), and call-cc (cf. SCHEME), - (return code of current outermost expression), provide a handle on the current status of the interpretation. However, continuations and environments are not represented as explicit first-order LISP objects.

One can for example not create or manipulate continuations in a regular LISP the way one does in 3-LISP. In 3-LISP the continuation of the interpreter is a first-order data object. The continuation is represented by a lambda-expression which can be manipulated by reflective functions. In a regular LISP, continuations cannot be dismantled. Continuations are only accessible by a limited number of special reflective functions.

A 3-LISP implementation of the condition signallers and condition handlers of ZetaLISP discussed in chapter II would look as in figure 12. The reflective function condition-bind evaluates its argument "body" in an extended environment, in which some condition-handlers are bound. The reflective function signal-condition calls the handler of the condition that is signalled. The rest of the object-computation is abandoned (it might possibly be reactivated explicitly in the handler).

```
(define-REFLECT condition-bind
   (list-of-condition-name-handler-pairs body &optional env cont)
      (eval body (append list-of-condition-handler-pairs env) cont)))

(define-REFLECT signal-condition (condition-name args &optional env cont)
   (funcall (binding condition-name env)
            args))
```

Fig. 12. Implementing conditions in a LISP with a reflective architecture.

The 3-LISP implementation of this exception handler is explicit, elegant and modifiable. It is also possible to define in 3-LISP the catch and throw functions of MacLISP, or to implement the fexpr functions of MacLISP, or even to invent exception-handling constructs which cannot be realised by any of the reflective facilities provided by the current LISP dialects.

Since the possibilities for reflection are open-ended in a reflective architecture, one should be careful in the design of this power (cf. dangers of reflection discussed in chapter II). Several solutions can be adopted. An architecture might for example only support weaker forms of reflection (cf.

chapter V). Or it might promote the positive uses of reflection, e.g. by means of libraries of typical reflective computations. Nevertheless, such a language would still have important advantages over regular languages. The whole issue of reflection remains perspicuous, as opposed to being handled in an ad hoc way. The particular style of programming which supports the separation and expliciteness of reflective computation is still preserved.

6. Conclusions

This chapter discussed a procedural, rule-based and logic-based example of a language with a reflective architecture. The examples were adapted from existing implementations.

These examples demonstrate that reflective architectures provide more elegant and more flexible facilities for reflection than an ad hoc selection of reflective constructs. Most importantly, they put forward reflection as a new paradigm for building and studying computational systems.

Chapter IV abstract the examples of this chapter to a general structure. It discusses the technical problems that have to be faced when building a reflective architecture and discusses techniques to solve these.

NOTES

[1] Actually 3-LISP is much more complex and has many more interesting ideas in it than what is illustrated here. The 3-LISP example is oversimplified in order to strenghten its illustrative power. The interested reader may find out about these other ideas in (Smith,1982), (Smith and Des Rivieres,1984) and (Des Rivieres and Smith,1984).

[2] The notion of an "impasse" in the inference process was introduced by SOAR (Laird, Rosenbloom and Newell,1984). The SOAR interpreter recognises four specific types of impasses which trigger reflection: a tie, a conflict, a no-change and a rejection. These really represent states in which the inference process gets stuck. It might be a little inapproriate to import this term to discuss reflection in TEIRESIAS. The events that trigger reflective computation in TEIRESIAS are the search for a goal and the making of a conclusion. These events not really represent a problem situation for the inference process.

CHAPTER IV

How To Build a Reflective Architecture

1. Introduction

Based on the examples in chapters II and III, we can see the general structure of a language with a reflective architecture. A language with a reflective architecture is a language in which computational systems are able to access and manipulate causally connected representations of themselves during computation. In order to build such a language, three problems have to be faced:

(i) the self-representation of the system,

(ii) programming reflective computation,

(iii) the causal connection link.

This chapter discusses each of these problems in detail and presents techniques to solve them.

2. The Self-Representation of a System

2.1. Introduction

First, the interpreter of the language has to be able to construct an explicit representation of a system and its current status. We have called this the **self-representation** of a system (cf. chapter I). It is on the basis of its self-representation that a system is able to reflect, i.e. to reason about itself and support actions upon itself.

The 3-LISP interpreter, for example, is able to construct an explicit representation of the process that is evaluating a piece of program. This self-representation consists of

(i) the code of the program,

(ii) an explicit representation of the program of the interpretation process (i.e. a circular interpreter), and

(iii) an explicit representation of the data of the interpretation process: two lisp-variables called "env" and "cont", respectively bound to the current environment (list of bindings) and current continuation of the interpretation.

By accessing these variables, a function is able to reason about itself and to modify its own behavior during its computation.

The interpreter of languages such as F.O.L. and META-PROLOG provides every theory with a meta-theory it can access. The meta-theory of a theory incorporates

(i) an explicit representation of the interpreter. (The circular F.O.L. interpreter is represented by the theory called META which is included in the meta-theory of every theory.)

(ii) an explicit representation of the object-level theory (i.e. of the facts and clauses that it contains).

So a language with a reflective architecture has to be able to build a representation of any system the interpreter is running. This problem has two subproblems. First, the language has to be able to generate information about a system and its current state of computation. And second, the language has to be able to construct an explicit representation of this information, such that it becomes accessible (i.e. becomes data) to the system itself. The next two sections study these problems.

2.2. The Meta-theory of the Language

A language with a reflective architecture has to be able to generate information about a system and its current state of computation. A computational system is defined by means of a program consisting of sentences.

Consequently, the language-interpreter has to ground the self-representation of a system X on the sentences that make up the program of X. Further, the computation of X is also determined by the run-time environment in which its sentences are interpreted, such as the status of the data and the history of the computation that preceeded.

So the self-representation of a system can be realised by incorporating in the interpreter a mechanism to generate information about a sentence used in a particular environment of computation. The implementation of this mechanism implies that the interpreter incorporates information about the meaning of the "generic" sentences of the language. Because this represents the information a language with a reflective architecture incorporates about itself, we call this the **meta-theory** of the language.

For example, assume we are building a variant of the language PROLOG which has a reflective architecture. When in a PROLOG program, a sentence like

```
man(X) :- father(X,Y).                    (i)
```

occurs, the language-interpreter we are building has to be able to generate some information about this piece of program. This means that the interpreter has to incorporate information about the meaning of the generic sentence

```
X :- Y.                                   (ii)
```

in order to be able to build an explicit self-representation of the specific instantiation (i). The information about (ii) for example states that (ii) can be used in a proof: if you want to prove that X is true, you can do this by proving that Y is true. We can encode this piece of meta-theory of the language in a "provable" predicate (see figure 13). The provable predicate of a logic-based language with a reflective architecture represents the language's meta-theory, or the theory it has about its sentences.[1]

```
provable(true).
provable(A,B) :- provable(A),provable(B).
provable(X) :- clause(X,Y),provable(Y).
```

Fig. 13. The provable predicate embodies the meta-theory of a logic-based language with a reflective architecture.

The first clause states information about sentences (here called terms) of the format

```
true
```

It specifies that these terms (actually there is only one) are always proved. The second clause tells something about the generic term

```
A,B
```

where A and B are terms again. It states that such a term can be proved by proving terms A and B independently. The third clause tells something about the generic term

```
X :- Y
```

where X and Y are terms. It states that such a term can be used to prove the term X, by proving the term Y.

This meta-theory can be used to generate information about a particular PROLOG program. E.g. we can generate information about the clause (i) above. By unification of the variables in the generic terms of the provable predicate to actual terms we can deduce

```
provable(man(X)) :- provable(father(X,Y)).
```

which says that the term

```
man(X)
```

can be proved by proving the term

```
father(X,Y)
```

For languages other than first order logic, specifying a meta-theory for the language is the main problem encountered when introducing a reflective architecture. Therefore, few non first order logic languages with a reflective architecture have been built. Finding a meta-theory for LISP was also the most difficult part in Smith's effort to build 3-LISP. Smith first had to straighten the semantics of LISP, such that the meaning of programs could be uniquely determined from the meaning of their (sub-)sentences (for regular LISP this poses problems because of quote). Only after that was he able to introduce a reflective architecture.

2.3. The Reference Mechanism

A language with a reflective architecture not only has to be able to generate information about a system it is running, but it also has to be able to construct a representation containing this information such that it becomes accessible. Three aspects are important here.

First, it is crucial that the self-representation is encoded in the format of normal data handled by the system. Only this way can a computational system actually manipulate the data representing itself, the way it manipulates object-data. This means that the language has to be able to view sentences in the language both as code to be executed and as data for reflective computation.

Second, although object data and reflective data should have the same format, it must still be possible to syntactically distinguish between an occurence of a sentence E as a normal sentence and an occurence of sentence E as the subject of a reflective sentence. Failure to respect this distinction produces paradoxes and confusion. For example the English language sentence

He said I'll do the job.

has a different meaning than the sentence

He said "I'll do the job".

We need clear markers between object-level and reflective level sentences. Most of the time, this problem is solved by **quoting** sentences in the language when they occur as the subject of a reflective sentence.

And third, the interpreter not only needs to incorporate a quoting mechanism for making **reflective sentences** (or sentences in which one specific other sentence is named), but it actually has to incorporate a **reference mechanism** for making **reflective representations**. The interpreter cannot always quote the sentence a reflective sentence is about. This means that the architecture has to provide an abstraction mechanism for using variables that range over sentences.

This means that we need a mechanism for making sentences which convey information about other sentences without actually naming them. A reference mechanism is realised by (i) providing a mechanism for making representations of sentences in the syntax of the language, and (ii) incorporating a method for retrieving the **referents** of these representations, i.e. all the sentences of the system a reflective representation applies to, in the interpreter of the language.[2]

E.g. in a PROLOG with a reflective architecture, the language provides a way to construct sentences in which parts of the sentence are a representation of other sentences. During computation, the interpreter tries to match these reflective representations with actual sentences, in order to find the sentences for which this reflective sentence actually applies. The term

```
clause(X,Y)
```

is an example of a PROLOG reflective representation. The language interpreter matches the variables X and Y in this term with all pairs of terms (A,B), which fulfill the constraint that the sentence

```
A :- B.
```

is part of the program. This interpreter is said to provide a **format-directed** reference mechanism. The referents of a reflective representation are retrieved on the basis of the syntactical format of the sentences.

The TEIRESIAS system incorporates an alternative reference mechanism, called **content-directed** invocation (Davis,1982). The reflective rules of TEIRESIAS refer to rules by describing them and effect this description by direct examination of the content of other rules. Consider the following reflective rule

```
IF 1) there are rules which do not mention the current goal in
      their premise
   2) there are rules which mention the current goal in their premise
THEN it is definite that the former should be used before the latter
```

The sets of rules this rule refers to is computed at run-time. The reflective rule examines at run-time (when it is executed) the source-code of all the rules of the rule-base to separate those that mention the current goal in their premise from those that do not mention the current goal in their premise.

The F.O.L. system (Weyhrauch,1980) presents yet another reference mechanism, called **semantic attachment**. In F.O.L. the linguistic structures that make up a (meta-)theory are attached to their "simulation structures" (i.e. referents) in the theory. The F.O.L. programmer has to specify explicitly which terms in the meta-representations refer to which object-level aspects. For example, the individual terms of a meta-theory are attached to the terms of the theory they represent, and the inference rules of a meta-theory (the theory META) are attached to the actual implementation of the inference mechanism of the F.O.L. system. For example, one can specify that the (meta-) term

```
'innocent(R,S)
```

is semantically attached to all terms that match

```
innocent(R,S)
```

in the theory. By default the predicate

```
provable
```

is attached to the implementation of the proof procedure of F.O.L. (which is a LISP program).

3. Programming Reflective Computation

3.1. Introduction

The second problem in the construction of a reflective architecture is called the problem of **programming reflective computation**. A system should be able to access and modify its self-representation, such that it can reason about and act upon its own computation. It must be possible to conditionally halt the object-computation and shift to a reflective level in order to access and manipulate the representation of that computation, and vice versa, to conditionally halt reflective computation and shift back to the affected object-level computation. Two aspects play a role here:

(i) what triggers a shift of levels,

(ii) what kind of computation is pursued after a shift of levels.

These two problems are discussed below.

3.2. Level-Shifting Triggers

Several mechanisms for triggering a shift of levels exist. The trigger to go from object-computation to reflective computation is in most languages different from the trigger to go from reflective computation back to object-level computation.

Reflective computation is in most languages, e.g. FOL (Weyhrauch,1980) and 3-LISP (Smith,1980), explicitly controlled (or triggered) by the systems themselves. A programmer has the possibility to specify special commands in a program which, when interpreted, bring the computational activity of a system to a reflective level. E.g. in 3-LISP, a system shifts control to the reflective computation when a reflective function is called.

SOAR (Laird, Rosenbloom and Newell,1984) presents a mechanism for implicit reflection. Reflective computation is triggered by an impasse during problem solving. When the computation cannot proceed, the SOAR interpreter sets up reflective computation which will try to resolve the impasse. The system TEIRESIAS, on the contrary, systematically reflects before trying to achieve a new goal.

There also exist several mechanisms to return from the reflective computation to the object-computation again. In some languages, e.g. in F.O.L. or TEIRESIAS, a system returns to the object-computation once the reflective computation is done. In other languages, e.g. in SOAR, the object-computation is resumed the moment the impasse is dissolved. A reflective function in 3-LISP explicitly prescribes when the object-computation should be continued.

3.3. Computation after a Level Shift

A reflective architecture not only provides triggers for shifting between object-computation and reflective computation, but it also determines what kind of computation is pursued after a level-shift happened.

Several designs can be adopted for going from the object-computation to reflective computation. 3-LISP programs prescribe the computation that has to take place after a shift to the reflective level. A 3-LISP program prescribes the reflective computation that has to take place just like it prescibes the object-computation that has to take place. It simply calls a reflective function, which is defined like an object-level function (except for the key-word "reflect").

Other languages implement a notion of default reflective computation which can be extended or overridden for a specific computational system. E.g. TEIRESIAS incorporates two types of reflective code (cf. chapter II)[3]

(i) domain independent code (i.e. the default reflective code),

(ii) domain dependent code (i.e. explicitly provided code).

TEIRESIAS incorporates default reflective rules such as the rule

```
IF 1) there are rules which do not mention the current goal in
      their premise,
   2) there are rules which mention the current goal in their premise,

THEN it is definite that the former should be done before the latter.
```

A particular system may modify or extend this default reflective code. For example, a system can interfere with this reflective rule, by stating the domain-dependent reflective rule

```
IF there is a rule which mentions peril danger in its conclusions and
   which mentions in its premise a previous organism which may be
   the same as the current organism
THEN it is definitely (1.0) a useful rule
```

TEIRESIAS is built such that user-supplied reflective rules have priority over default reflective rules.

Another way the default reflective computation can be modified, is by stating reflective rules about the default reflective rules. The following example illustrates such a reflective rule

```
IF there is a rule which mentions in its conclusion the
   definite utility of a rule, and
   which mentions in its premise the current goal
THEN this rule is definitely useless.
```

Different designs can also be adopted when returning from the reflective computation to the object-computation. In most architectures the object-computation is continued taking into account the current status of the self-representation existing at the reflective level. E.g. in TEIRESIAS or SOAR, the object-computation that is continued is determined by the status of the self-representation. 3-LISP presents an exception to this rule. If a reflective function of 3-LISP has to reactivate the object-level computation, it not only has to say that this should happen, but it also has to say how this should happen. Going down happens by applying the continuation of the lower-level computation or by calling the evaluator explicitly.

4. The Causal Connection Requirement

4.1. Introduction

The third and final problem to be faced in the construction of a reflective architecture is the causal connection requirement (Smith,1982). The self-representation of the system that is manipulated at the reflective level has to be connected to the system itself in such a way that when the reflective computation is resumed, the object-level computation continues with a program and a run-time environment that is affected by the changes made to the self-representation. Vice versa, the self-representation of a system always has to be connected with the actual state of the system, such that whenever something changes to the state of the system, this change is reflected in the self-representation of the system.

For example, if a 3-LISP function decides to consult its self-representation, this representation has to be accurate. Figure 14 shows a function foo that consults its self-representation to check whether the variable "bar" is bound. At that moment it is important that the variable "env" in the reflective function "boundp" is bound to the environment of the moment the function boundp was called.

```
(define-REFLECT boundp (symbol &optional env cont)
       (funcall cont
                (not (null
                        (mapcan
                           '(lambda (one-binding)
                                (if (eq (car one-binding) symbol)
                                    (list T))
                           env)))))

(define-SIMPLE foo (...)
    ...
    (if (boundp bar) ...)
    ...)
```

Fig. 14. The importance of upward causal connection.

Vice versa, if during reflective computation the self-representation is changed, then the actual computation that is continued after reflection should be in accord with the new self-representation. Figure 15 illustrates a function foo that calls a reflective function which halts the computation (it does not reactivate the object-computation) This modification actually affects the computation of the function foo: the forms (i) in the program of foo are not evaluated, and foo returns as result the string "halted".

```
(define-REFLECT halt (&optional env cont)
   "halted")

(define-SIMPLE foo (...)
   (prog (...)
      ...
      (halt)
       ...
       ...  } (i)
       ...
   ))
```

Fig. 15. The importance of downward causal connection.

In general, when designing the causal connection system of a reflective architecture two problems have to be solved. First some **monitoring mechanism** has to be developed. The purpose of the monitoring mechanism is to check whether some changes occur to either the system or its self-representation.

The second problem to be solved is the development of a mechanism in the interpreter which is activated when such a change occurs. We call this the **representation maintaining mechanism** of the reflective architecture. The representation maintaining mechanism is some sort of consistency maintenance mechanism. Its purpose is to realise the effects a change to either the system or its self-representation should cause in the other one. So actually this mechanism takes care that the reflective representations discussed in section 2.3 are maintained. It maintains the reference-links between the reflective representations and the system.

4.2. Examples

The easiest and most common technique to solve the causal connection problem is to make the language operational by means of a **meta-circular interpreter**. A meta-circular interpreter is

(i) an explicit representation of the interpreter in the language itself,

(ii) which is also actually used to run the language.

The idea is that if we want to give systems a representation of aspects of their own execution process, the easiest solution is to make the execution process itself a computational system in the language. So the explicit representation of a system (in terms of an explicit representation of its interpretation) is actually used to run the system. A necessary condition for a meta-circular representation 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.

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 an interesting consequence of a meta-circular interpreter is that there is not really a representation mechanism needed (because there are not two distinct representations). There only exists one representation which is both used to implement the system and to reason about the system.

Computation in a language with a meta-circular interpreter can be viewed as an infinite tower of systems all computing about the level below (Smith,1982). The lowest level does computation about the actual problem domain. The other levels manipulate causally connected representations of the level below.

The 3-LISP interpreter, for example, is (virtually) an infinitely stretching tower of read-eval-print loops. A production rule system with a meta-circular interpreter is (virtually) an infinite hierarchy of problem-spaces, where the goal in each problem-space is to help forward the computation of its super-space. Similarly, every theory in a logic programming language with a meta-circular interpreter would (virtually) have its meta-theory.

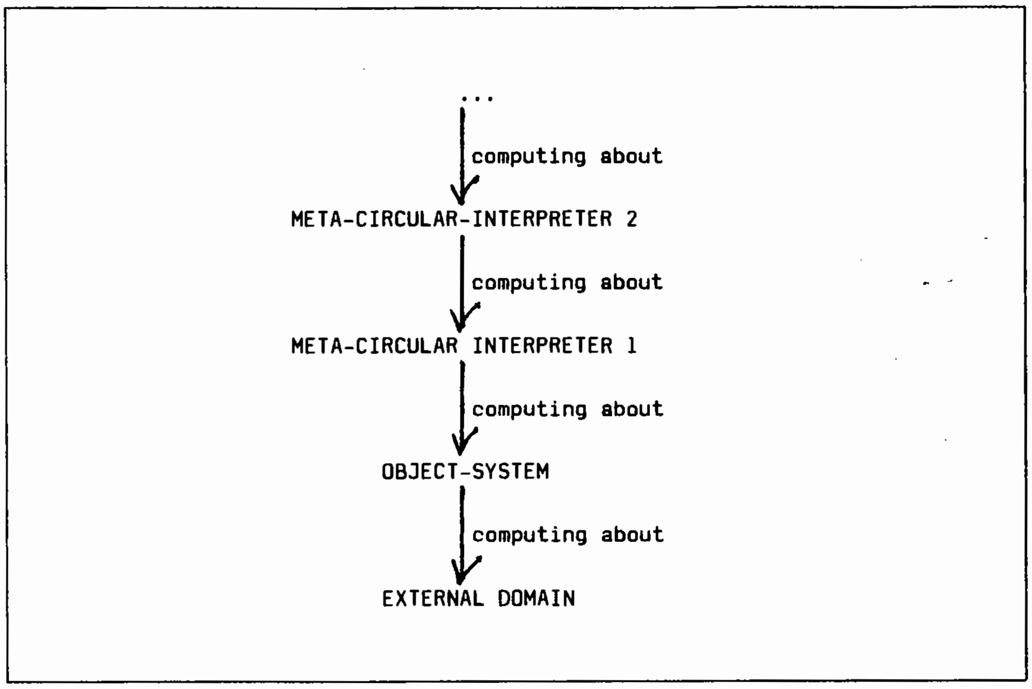


Fig. 16. Computation in a language with a meta-circular interpreter virtually happens at an infinitely stretching number of levels.

It is technically possible to implement this infinity because a specific application only uses a finite number of levels, which means that one only needs to construct the number of meta-circular interpreters that are actually used. However, a necessary condition to let a language actually operate by means of a meta-circular interpreter is that there exists a second interpreter written in another language which is able to interpret the meta-circular one (in order not to fall into a loop of meta-circular interpreters interpreting meta-circular interpreters). Two interpreters are necessary, but also sufficient: we can generate meta^n interpreter-levels by recursively using the meta-circular interpreter.

An efficient technique that can be used to implement a meta-circular interpreter is discussed in (Des Rivieres and Smith,1984). We will refer to this technique as **reification**. Reification consists of reifying (or making explicit and thus available) the data-structures of the interpreter to the program that is running whenever it asks for it (explicit copies of the real ones), and reinstalling these data-structures in the object-computation when the reflective computation has finished (installing the copies into the real interpreter again). This technique guarantees that at all times only the number of interpreter-levels that is actually needed exists.

It is realised by incorporating in the second interpreter (the real one) a mechanism that is able to dynamically create and delete circular interpretation levels. First the second interpreter has to be modified such that it is able to recognise reflective code in the program that it is interpreting. The interpreter starts with one level.

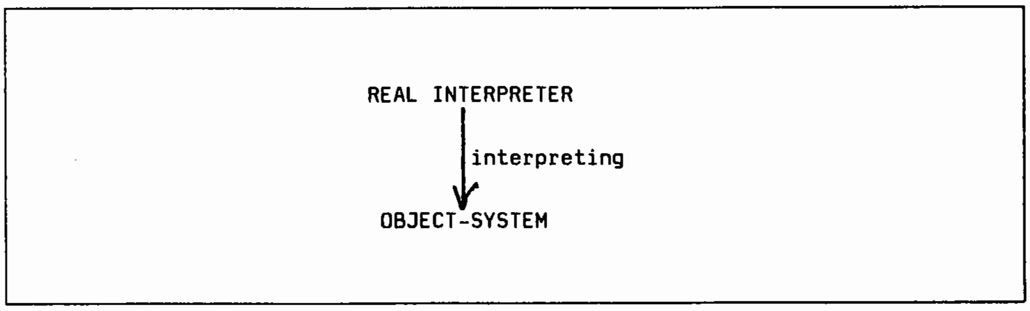


Fig. 17. Initial status of a meta-circular interpretation.

Whenever the interpreter identifies some reflective code in the code of the object-system program, the interpreter has to create an explicit interpreter-level. It has to initialise this meta-circular-interpreter with (explicit) copies of its own status (e.g. its continuation, current-expression, env). After that it has to reinitialise its own status. The new program that it has to interpret is the reflective code. The new values of the continuation and environment of the second interpreter vary in different designs. The reflective code will now be interpreted by the second interpreter, while the object-code is interpreted by the meta-circular interpreter.

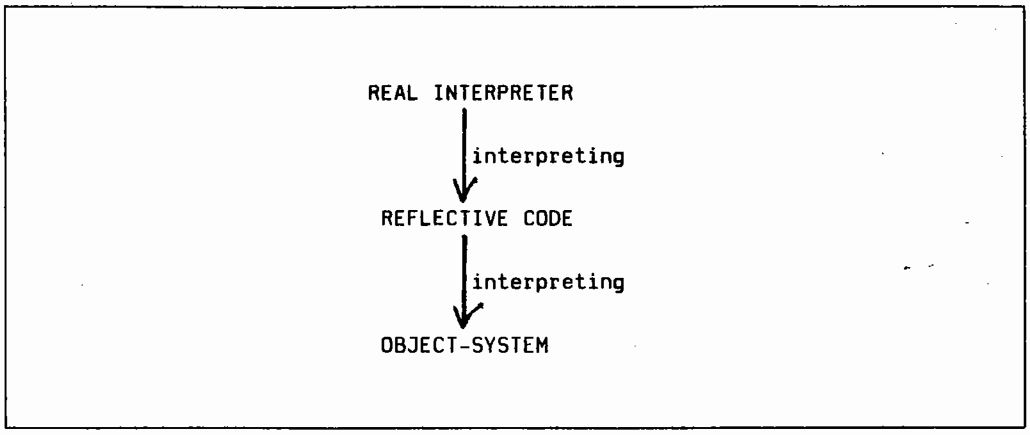


Fig. 18. The number of levels is expanded by the presence of reflective code.

This expansion of levels is performed whenever it is needed. For example, when during interpretation (by the second interpreter) of the reflective code, again reflective code is encountered (i.e. this is actually reflective^2 code), a third interpreter-level will be created in a similar way. The number of levels is schrunk whenever the second interpreter has finished the interpretation of the code of the level below. At that moment the second interpreter adopts the status of the top-most meta-circular interpreter and throws the latter one away.

F.O.L. adopts another technique. The causal connection downward is in F.O.L. implemented by means of the reflection principles discussed in chapter II. For example, the reflection principle

$$\begin{array}{ll} \texttt{(In meta-T)} & \dfrac{\texttt{theorem(W)}}{\texttt{W}} \\ & \\ \texttt{(In T)} & \end{array}$$

takes care that new theorems deduced in the meta-theory, are exported to (or asserted in) the object-level theory. The semantic attachment technique (discussed previously in this chapter) handles the causal connection upward. The meta-theory always has an accurate representation of its object-theory because the meta-theoretical terms are attached to the aspects of the object-theory they represent.

5. Reflective Languages

When the interpreter of a language with a reflective architecture is meta-circular, we can also view this interpreter as a program in the language. This means that this system can also be given a reflective behavior. When the interpreter for language L is also a reflective system in L, we speak of a **reflective language**. When the language itself reflects this should have a global impact upon the computation of all systems implemented in the language. In order to make reflection by the language possible the following three constraints have to be fulfilled:

(i) the language has to run by means of a meta-circular interpreter (which means that there is also a second interpreter simulating a tower of default-interpreters), and

(ii) this meta-circular interpreter has to be accessible and modifiable in a global way (instead of only being modifiable from within the interpretation of one program), and

(iii) the meta-circular interpreter and the second interpreter not only have to cause the same behavior, but they should also be causally connected to each other.

Actually, (iii) states that the second interpreter of the language has to be sensitive for a change to the global meta-circular interpreter, such that whenever reflection modifies something to the meta-circular interpreter, the whole behavior of the language changes accordingly. The technique that is usually applied to bottom out the recursion of a meta-circular interpreter is that only reflective programs are interpreted by the meta-circular interpreter, while non-reflective code is immediately interpreted by the second interpreter. Consequently, a technique that can be used to bottom out the recursion in a reflective language is that there is one extra interpretation step for all programs in the language: non reflective programs need two steps instead of one, reflective programs need three steps instead of two.

A reflective language represents an even more powerful tool than a language with a reflective architecture. Examples of things which can be realised in a reflective language are:

- To change the semantics of the syntax of the language during computation. For example, a reflective logic programming language could change the meaning of the implication sign such that the interpretation of all programs in the language is affected.
- To improve the control aspect of the interpreter of the language during computation. For example, a reflective rule-based language could "learn" strategies to interpret rule-based programs in a more effective way.

Although most languages with a reflective architecture are implemented by means of a meta-circular interpreter, it is not clear from the literature whether these languages are also actually reflective languages. From what is documented, we conclude that the languages cited until now (3-LISP, F.O.L., METAPROLOG, TEIRESIAS) are not reflective languages. What is missing in the architecture of these languages is the requirement (iii). The meta-circular interpreter and the second interpreter are not causally connected to each other. Therefore, we call the language 3-KRS, discussed in chapter VI of this dissertation, the first operational reflective language.

6. The Cost of a Reflective Architecture

Including a reflective architecture in a language involves three costs:

(i) The cost of setting up the reflective architecture,

(ii) The cost while interpreting the language,

(iii) The cost of reflective computation.

The initial cost of a language with a reflective architecture is the cost of designing and implementing the reflective architecture. This cost is probably high at the moment. However, it will become less relevant as reflection becomes better understood and techniques for building reflective systems become established. Note that designing a new language may be facilitated by a reflective architecture. (Smith,1982) shows how the kernel of the LISP language can be substantially reduced by adopting a reflective language. Several functions (such as "defmacro") no longer have to be primitive functions of the language.

As regards the runtime costs, languages with a reflective architecture are not necessarily inefficient languages. The self-representation of a system that is running has to be maintained. However, it only needs to be constructed when needed, i.e. when it is actually used in reflective computation. This means that when the reflective facilities of the language are not exploited by a system, the run-time cost of a reflective architecture is almost equal to that of a non-reflective architecture. The only additional computation is a test that checks at the appropriate moments whether a system wants to make use of the reflective level or not.

The cost of reflective computation has two components: (i) the effort of the programmer and (ii) the effort of the machine. Programming in a reflective architecture does not necessarily require too much (system-specific) knowledge. The self-representation provided in current reflective architectures is designed to suit the interpreter that has to realise it (and its causal connection). However, programming reflective systems could be substantially facilitated by self-representations that are designed to suit the programmer that has to use it. Another way programming reflective systems can be improved is by providing the programmer with libraries of commonly used reflective computations from which he can pick a specific type of reflective computation whenever needed.

The other component in the cost of reflective computation is the effort the interpreter has to make when a system makes use of its reflective abilities. Computing at run-time the object-level computation that has to be performed is of course more costly than having a fixed scheme of computation. This difference in cost is similar to the difference between evaluating code and executing compiled code.

A solution here might be that, either reflective computation is exceptional, or, when it is not exceptional, it becomes compiled into the actual interpreter of the language. Some reseachers have been looking for techniques to compile reflection. For example, (Genesereth and Smith,1981) describes an attempt to recognise deterministic reflective behavior and replace it with algorithmic code. (Van Harmelen,1986) gives a survey of possible techniques to improve the efficiency of meta-level architectures and reflective architectures.

7. Conclusions

Three problems have to be solved when building a language with a reflective architecture:

(i) The problem of the **self-representation of a system** implies that the interpreter of the language has to be able to generate a representation of a system and its state of computation.

(ii) The problem of the **programming reflective computation** implies that the architecture has to define the conditions under which a shift between object-level computation and reflective computation happens.

(iii) The problem of the **causal connection link** implies that the self-representation of the system that is manipulated at the reflective level has to be causally connected to the system itself.

This chapter analysed these problems and presented techniques to solve them. The next chapter studies the relation between the specific techniques that is chosen and the possibilities for reflection that result.

NOTES

[1] The meta-theory of a language is in almost all of the existing reflective architectures embodied in a meta-circular interpreter.

[2] These problems are reflected in Smith's first and second factor. (Smith,1982) argues that when talking about a specific reflective architecture, one should always specify an account of the following two factors for a sentence in the language

(i) the **first factor** also called functional role or procedural consequence or immediate effect is what a sentence returns (what results from a computation, inference, deduction, or message passing, i.e. the computer science notion of interpretation) and

(ii) the **second factor** also called significance or content, is what sentences are about (model theory, designation, logic's notion of interpretation)

Point (ii) is important to know the referent of a self-representation. (i) is important to know which aspects of the referent a self-representation represents (see chapter VI).

[3] It is therefore that some authors distinguish three levels of information in a computational system which is embedded in a reflective architecture:

(i) representations of things in the outside world,

(ii) representations about those representations, and

(iii) representations about representations in general.

CHAPTER V

Design Issues

1. Introduction

The previous chapter elaborated on the construction of a reflective architecture. Not all reflective architectures that are built provide equal possibilities for reflection. The reflective facilities of a language with a reflective architecture depend upon the design choices that were made in order to solve the three problems discussed in the previous chapter:

(i) what representation a system has of itself,

(ii) how the reflective computation of the system is programmed, i.e. under what conditions a system will manipulate its self-representation and what manipulations it will perform,

(iii) the causal connection link that exists between the system and its self-representation.

This chapter discusses how reflective architectures may vary along these three dimensions. It does not advocate particular design choices, but argues that a choice along one dimension has to be made with a particular kind of reflective facility in mind. The three dimensions define a space of alternative designs reflective architectures can explore. However, the reflective architectures that have already been built, are grouped in a few areas of this space.^{1}

2. Theory Relativity of a Reflective Architecture

2.1. Introduction

Languages with a reflective architecture give systems the ability to reflect by providing them at run-time with a causally connected self-representation they can access and manipulate. This self-representation determines what computation the system is able to perform about itself and what modifications it is able to make to itself.

A specific self-representation defines a terminology the system can use to specify reflective computation. It fixes the set of terms a system can use to specify reasoning about and acting upon itself. Because this terminology is directly determined by the meta-theory of the language, a reflective architecture is said to be **theory relative** (Smith,1982)[2].

2.2. The Terminology of the Meta-Theory

Every representation defines a certain **terminology** to talk about the entity(ies) it represents. For example, the position of a robot-arm can be represented by a triple of coordinates. This representation makes it possible to ask for the x-coordinate of the arm, or to state that the x-coordinate of the arm has a certain value. However, since this representation does not represent whether the hand of the robot-arm is open or closed, it cannot be refered to. It is not possible to ask whether the hand is open, nor can it be stated that it is open.

The same holds for the self-representations of a reflective architecture. A self-representation defines a terminology to refer to aspects of the computational system it represents. Since a self-representation is built on the basis of the meta-theory of the reflective architecture, the terminology of the self-representation is the terminology of the meta-theory. The terminology introduced by the meta-theory of a reflective architecture varies a lot. For example, an explicit representation of the operational aspect of systems in LISP might range from

```
(define meta-circular-interpreter-1 (expr &optional env cont)
   (eval expr env cont))
```

to a more extended version

```
(define meta-circular-interpreter-2 (expr &optional (env ()))
   (cond
      ((null expr) NIL)
      ((numberp expr) expr)
      ((eq expr T) expr)
      ((symbolp expr)(binding expr env))
      ((eq (car expr) 'quote) (cadr expr))
      ((primitive-function-p (car expr))
          (apply (car expr)
                 (make-list-of-evaluated-args (cdr expr) env)))
      (T (eval (definition-of (car expr))
               (lexical-make-env
                  (definition-args-of (car expr))
                  (cdr expr)
                  env)))))
```

or to one that makes explicit all machine actions (from Sussman,1982)

```
(defpc meta-circular-interpreter-3
   (save env)
   (save unev)
   (save argl)
   (save fun)
   (save retpc)
   (goto eval-dispatch)


(defpc pop-return
   (restore retpc)
   (restore fun)
   (restore argl)
   (restore unev)
   (restore env)
   (goto (fetch retpc))
```

```
(defpc eval-dispatch
   (cond ((self-evaluating? (fetch exp))
          (assign val (fetch exp))
          (goto pop-return))
         ((quoted? (fetch exp))
          (assign val (text-of-quotation (fetch exp)))
          (goto pop-return))
         ((variable? (fetch exp))
          (assign val
                  (get-variable-value (fetch exp)
                                      (fetch env)))
          (goto pop-return))
         ((lambda? (fetch exp))
          (assign val
                  (make-procedure (fetch exp)
                                  (fetch env)))
          (goto pop-return))
         ((conditional? (fetch exp))
          (assign unev (clauses (fetch exp)))
          (goto evcond-pred))
         ((no-operands? (fetch exp))
          (assign exp (operator (fetch exp)))
          (assign retpc apply-no-args)
          (goto meta-circular-interpreter-3))
         ((appliciation? (fetch exp))
          (assign unev (operands (fetch exp)))
          (assign exp (operator (fetch exp)))
          (assign retpc eval-args)
          (goto meta-circular-interpreter-3))
         (t (error "unknown expression type -- eval"))))
```

The terminology introduced by meta-circular-interpreter-1 is rather limited. It makes it possible to refer to four aspects of the interpretation of a LISP program: the evaluator program, the program itself, the environment and the continuation. A system which has this interpreter as representation of itself, may for example refer to its own environment, or it may act upon its own interpretation, as in

```
(define variant-on-meta-circular-interpreter-1 (expr &optional env cont)
   (do-something-with-the-result
      (eval (do-something-with-the-input expr))
            env)
            cont)))
```

This is a variant on the interpreter, which will do something before and after the (default) evaluation of an expression. The self-representation of meta-circular-interpreter-1 only makes it possible for a system to reason about and act upon the LISP-evaluator as a whole. It does not make it possible to analyse how the evaluator works, or to modify internal aspects of the evaluator.

On the other hand, meta-circular-interpreter-2 already introduces a more sophisticated terminology to refer to the operational aspect of a computational system in LISP. This self-representation makes explicit some of the internal aspects of the LISP-evaluator. It makes it for example possible for a system to ask questions about the evaluation of an expression of a particular syntactical type. It also allows more fine grained modifications to the LISP-evaluator, such as

```
(define variant-on-meta-circular-interpreter-2 (expr &optional (env ()))
   (cond
      ((null expr) NIL)
      ((numberp expr) expr)
      ((eq expr T) expr)
      ((symbolp expr)(binding expr env))
      ((eq (car expr) 'quote) (cadr expr))
      ((primitive-function-p (car expr))
          (apply (car expr)
                 (make-list-of-evaluated-args (cdr expr) env)))
      (T (eval (definition-of (car expr))
               (dynamic-make-env
                  (definition-args-of (car expr))
                  (cdr expr)
                  env)))))
```

This variant modifies the interpreter such that it has dynamical scoping. In the original interpreter a function-definition was interpreted in the environment containing only the bindings made by the arguments of the function-call. The function lexical-make-env used in the original interpreter was defined as follows

```
(define lexical-make-env (definition-args formula-args old-env)
   (cond ((null definition-args) nil)
         (T (add-binding (car definition-args)
                         (eval (car formula-args) old-env)
                         (lexical-make-env (cdr definition-args)
                                           (cdr formula-args)
                                           old-env)))))
```

In the variant interpreter, function-definitions are evaluated in the environment at the moment they are called, extended with the bindings made by the arguments of the function-call. The function dynamic-make-env adopted in this variant interpreter is defined as

```
(define dynamic-make-env (definition-args formula-args old-env)
   (cond ((null definition-args) old-env)
         (T (add-binding (car definition-args)
                         (eval (car formula-args) old-env)
                         (dynamic-make-env (cdr definition-args)
                                           (cdr formula-args)
                                           old-env)))))
```

Note that this variation cannot be made by means of meta-circular-interpreter-1. However, meta-circular-interpreter-2 does not make explicit the continuation of the evaluation (which is implicit in the recursion) as meta-circular-interpreter-1 does. Consequently, some variations on meta-circular-interpreter-1 will not be expressible in the terminology introduced by meta-circular-interpreter-2.

Meta-circular-interpreter-3, on the other hand, presents a self-representation of the operational aspect of systems in terms of machine concepts such as registers and memory-adresses. This makes it for example suited for analysing and modifying the time and space requirements of systems. However, it leaves implicit other aspects of the interpreter, such as its recursive behavior.

2.3. A Space of Variations

2.3.1. Introduction

The terminology of the meta-theory of a reflective architecture defines the aspects along which a system may analyse and modify itself. It determines a **space of variations** on itself a system is able to construct.[3] Although this space is infinite, it is always bounded. It is always possible to think of a variation that cannot be expressed in the finite terminology.[4]

The meta-theory a language has of itself (and consequently the self-representation a system has of itself) is never complete. It is an inherent property of representations that they necessarily contain less information than the thing they represent. Consequently there are always aspects or relations of the system which are implicit in the self-representation and which can thus not be modified. Consequently, when designing a reflective architecture, the problem is to choose the optimal self-representation for the kind of computation the language aims to support.

One of the most important questions that has to be considered when trying to choose a particular meta-theory is what aspect(s) of itself a computational system should be able to reflect upon. First there are aspects which can be extracted from the sentences that make up the program of the system (the default self-representation (cf. chapter IV)). Four such aspects can be identified: the syntax, semantics, deduction (or computation) and operation. In addition, the programmer may make arbitrary aspects of systems explicit. He may even make information explicit which is not implicitly present in the data, program or execution of the system, e.g. who created the system.

The following sections discuss the first four aspects. They show that meta-theories making explicit different combinations of these aspects result into entirely different facilities for reflection. The discussion is illustrated with examples from a logic language, but the same aspects can be identified for other languages as well. The next chapter will do it for an object-oriented language.

2.3.2. Syntax

Computational systems have a **syntactical aspect**. A language imposes a particular format upon programs and data. For example, the format of LISP are S-expressions, the format of logic are clauses and the format of rule-based systems are rules. The meta-theory of the reflective architecture may formalise this syntactical aspect of systems. Systems then have the possibility to reason about or manipulate the object-level syntax of the language from within the language itself. A reflective system can for example extend the syntax of sentences at the object-level by stating at the reflective level

```
        legal-sentence("P <=> Q.").
```

From that moment on, it is possible to use this new syntactic construct (although there is not yet a meaning associated with it).

2.3.3. Denotation

Computational systems have a **denotational aspect** (also called **semantical** aspect). A language has rules to relate the syntactical structures in the program of a system to objects and relations in a problem domain. The denotational semantics of a programming language formalises these rules by constructing functions that map programming constructs to their denotations. The meta-theory of a reflective architecture may incorporate this denotational aspect of the language.

This requires that the language has some notion of denotation, i.e. of dereferencing of syntactic structures and creating structures with specific referents. However, this is not the case for extensional logic programming languages (where all inference happens by applying deduction rules). Nevertheless, we can imagine a logic-based language in which reflective systems can affect the denotation of syntactical structures at the object-level by stating facts such as

```
equal-denotation(Patricia,Pattie).
```

which states that structures Patricia and Pattie have the same denotation. This would of course require that the interpreter actually uses such co-denotational structures interchangeably during deduction (e.g. in unification).

2.3.4. Deduction/Computation

Computational systems have a **deductive aspect** or a **computational aspect**. Usually the answers to questions are not explicitly represented in the data of a system, but must be deduced (in a declarative language), or computed (in a procedural language). Note that it is important that the denotational and deductive/computational aspect of a language are consistent.

The deductive aspect of a system can be subdivided in the inference theory and the control aspect. The **inference theory** of a declarative language defines the allowable inferences that can be made (i.e. the allowable changes to the data of the system, given the program of the system). The meta-theory may formalise this inference aspect of the language. Systems then are able to prove properties about their own inference theory (e.g. monotonicity, provability) or to extend it, as in (in italics)

```
provable(P) :- theorem(P).
provable(and(P,Q)) :- provable(P), provable(Q).
provable(P) :- clause(P,Q), provable(Q).
clause(P,Q) :- theorem("P <=> Q.").
clause(Q,P) :- theorem("P <=> Q.").
```

The new syntax construct created in section 2.3.2. can thus also be used in programs, as in

```
even(X) <=> divisible(X,two).
```

The inference theory of a declarative language defines the possible inferences made by a computational system, but does not say which ones should actually be made. This is the role of the **control aspect** of the language. There can be a sequential execution of the program, a goal-oriented execution of the program, a data-driven execution of the program, etc. The meta-theory of a reflective architecture may formalise this control aspect. Systems then have the possibility to reason about and experiment with different control strategies. For example,

```
goal(P) :- not(theorem(P)), clause(P,Q), goal(Q).
```

In a procedural programming languages the control and deductive aspects of a computation system cannot be disconnected. Programs specify at the same time what computation should happen and when it should happen. The term **computational aspect** is here used to refer to both aspects at the same time. If the meta-theory of a procedural language formalises this computational aspect, reflective systems are able to reason about an act upon their computation and its flow. Some LISP examples of functions doing exactly this were presented in chapter I, II and III.

2.3.5. Operation

Computational systems have an **operational aspect**. To actually realise a computational system it must be emulated by a computational medium. Most languages are emulated by an interpreter, written in another language (which is also emulated, for example, by machine-language). The meta-theory may formalise this operational aspect of the language. Systems then have the possibility to reason about how they are being executed.

For example, when the language is PROLOG and this PROLOG is interpreted by means of an interpreter written in PROLOG, then the meta-theory could incorporate a causally-connected representation of how this interpreter works. Systems can then reason about and act upon the PROLOG computation that is being performed while processing them. They can for example reason about their own performance or modify their own interpreter or extend the environment of their execution, etc.

3. Variations on Programming Reflective Computation

3.1. Introduction

When designing a reflective architecture, the second design choice to be made is how the reflective computation will be programmed, i.e. what the conditions are under which a system will manipulate its self-representation and what kind of reflective computation the system will perform when these conditions are fulfilled. Are there specific points in the interpretation of the system where the self-representation is examined? Does the system itself specify when it wants to reflect? Is the reflective computation continously active? Does the system or does the architecture decide what kind of reflective computation is pursued? Is there a notion of default reflective computation? How does the computation proceed after reflective computation?

Actually the decision to be made here is what the "interface" between object-computation and reflective computation looks like. Again there is a range of possible solutions, representing a second dimension along which reflective architectures vary. In order to make a choice along this dimension, the fundamental question to be asked is what the role of reflective computation is in the overall model of computation adopted by the language.

If reflective computation is a fundamental component of the model of computation of the language, i.e. if the well-functioning of the object-computation requires reflective computation, then the interface should be constructed such that systems systematically or constantly reflect. We say that such languages have an architecture for **implicit reflection**.

If reflection only serves to facilitate exceptional functionalities on the object-computation, i.e. if a system also runs without reflective computation, the interface should be designed such that reflective computation only takes place when the program of a system explicitly prescribes it. We say that such a language has an architecture for **explicit reflection**. It is possible that an architecture supports at the same time implicit reflection and explicit reflection.

3.2. Implicit Reflection

In an architecture for implicit reflection, reflective computation plays a crucial role. The reflective computation is systematically activated by the interpreter of the language. This means that there are "holes" in the normal interpretation process that remain to be filled by reflective computation.

If the interpretation process consists of the execution of tasks a_1 .. a_n, there exist one or more i, for which the action a_i is not defined in the normal interpreter (called "second interpreter" in chapter IV) . I.e. some a_i are necessarily a piece of reflective computation defined by the program of the system that is interpreted.

Since reflective computation is a crucial component of the interpretation of a system, the program of every system has to prescribe this reflective computation. Therefore the technique of default reflective computation is adopted in an architecture for implicit reflection. If the program of a system does not prescribe the reflective computation that should happen for the action a_i, the architecture executes a default reflective program.

TEIRESIAS (Davis,1982) is an example of a language with an architecture for implicit reflection. TEIRESIAS systematically reflects everytime a new goal has to be pursued by the interpreter. The purpose of this reflective computation is to compute the domain-dependent strategy to be used for the exploration of the search space for the goal. The default reflective computation that is adopted will use some general criteria to favor a specific strategy (cf. chapter II).

SOAR (Laird,Rosenbloom and Newell,1986) is another example of an architecture for implicit reflection. Although SOAR is also a rule-based system, the reflective computation in SOAR plays a different role in the interpreter. A computational system in SOAR reflects when the object-computation gets stuck in an impasse. The goal of the reflective computation that is activated at that moment, is to resolve the impasse. As soon as this is achieved, the reflective level is abandoned and the object-level computation is continued. Again default reflective programs to resolve particular impasses exist.

3.3. Explicit Reflection

In architectures for explicit reflection, reflective computation does not happen automatically. Reflection only takes place when the program of a system explicitly prescribes it. Computation normally takes place at the object-level. Whenever the program prescribes reflective computation, the system enters a break to temporarily do something at the reflective level.

Most languages who were initially designed without a reflective architecture, provide an architecture for explicit reflection. The reflective computation is not a crucial component of the interpretation process. Reflection serves more to support programming environment issues such as maintenance, modification, debugging, documentation, and use of computational systems.

3-LISP is an example of an architecture for explicit reflection. Reflective computation occurs in 3-LISP whenever a reflective lambda-expression is evaluated. A computational system in 3-LISP may not perform any reflective computation at all. Another example of an architecture for explicit reflection is F.O.L. The F.O.L. interpreter only jumps to a reflective level when the programmer explicitly states this.

4. Type of Causal Connection

4.1. Introduction

The third design choice to be made in the construction of a reflective architecture is how the causal connection requirement will be fulfilled. In order to establish a causal connection link between the self-representation and the aspects of the system it represents, a monitoring mechanism and a representation maintenance system has to be developed. The monitoring mechanism does not pose substantial design or implementation questions. The represention maintenance system, on the other hand, certainly represents one of the most difficult problems to be solved in the construction of a reflective architecture. Only a few solutions, of which none is very satisfactory, have been proposed until now.

The solutions that can be adopted fall into two classes called **declarative reflection** and **procedural reflection**. In both classes computational systems exhibit the behavior dictated by their self-representation. However, in the former this happens by means of an **implementation** link, while in the latter it happens by means of a **specification** link.

4.2. Procedural Reflection

In an architecture for procedural reflection, the causal connection relation between a system and its self-representation is realised by means of an implementation link. The causal connection between system and self-representation is guaranteed because the self-representation of a system is also used to implement it. So there is not really a representation maintenance problem. There only exists one representation which is both used to implement the system and to reason about the system.

The problem with this causal connection scheme is that a self-representation has to serve two purposes. Since it serves as the data for reflective computation, it has to be designed such 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.

Consequently substantial constraints are imposed on the format and the contents of a self-representation. It necessarily exists of a complete procedural representation of the operational aspects of a system, i.e. it necessarily is a meta-circular interpreter of the language. E.g. in 3-LISP, the self-representation necessarily consists of a function representing the evaluation procedure. It is not possible to represent the evaluator in terms of the input-output relations that it maintains, because the self-representation also has to be able to execute a system.

So, in an architecture for procedural reflection a self-representation can only be a straightforward representation of the entities of the interpretation process (such as the environment, the stack, and the interpretation-program). We will call these representations **reifications** of the entities in the implementation process. Reifications do not present the most interesting foundation for reflective computation that can be imagined (cf. issue of theory relativity discussed in section 2).

Examples of architectures for procedural reflection are 3-LISP and META-PROLOG.

4.3. Declarative Reflection

In an architecture for procedural reflection the computation of a system can only be controlled by giving a complete procedure for the implementation of the system. The motivation for research on architectures for declarative reflection is to build systems which accept explicit assertions about their object-computation and are able to continue their computation such that it fulfils these assertions.[5] These assertions could for example say that the computation of the system has to fulfill some time or space criteria. In these architectures the self-representation does not have to be a complete procedural representation of the system, it is more a collection of constraints the status and behavior of the system have to fulfill.

In an architecture for declarative reflection, the causal connection relation between the system and its self-representation is realised by means of a specification relation. The architecture has a notion of meta-representations, i.e. of representations denoting other representations. It provides means to create meta-representations, and to retrieve the referents of meta-representations. Further, the reflective architecture also incorporates a mechanism to maintain a meta-representational link. Whenever a meta-representation or its referent changes, the other one is updated such that the representational link is maintained.

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

An architecture for declarative reflection can be viewed as an architecture incorporating representations in two different formalisms of one and the same system. During computation the most appropriate representation is chosen. The implicit representation serves the implementation of the system, while the explicit representation serves the reasoning and acting upon the system.

Although architectures for declarative reflection are more difficult to implement than architectures for procedural reflection, they present a more interesting foundation for reflective computation. This is due to the fact that the self-representation of such an architecture only serves to describe the system, without being used as the foundation of its implementation. At this moment, a fullfledged architecture for declarative reflection has not yet been realised. There is even debate whether the problem is actually a solvable one (Des Rivieres,1986). Nevertheless, some important attempts are worth mentioning.

The goal of research on partial programs (Genesereth,1987) is to allow the specification of explicit constraints upon the object-level computation, which by means of the reflective architecture would be taken into account in the actual object-computation. The idea is that while doing computation about the external domain, the system can receive advice about the object-computation and can continue the object-level computation in line with this advice.

The GOLUX system (Hayes,1974) presents another attempt to build an architecture for declarative reflection. The reflective level of a GOLUX system describes (i.e. contains assertions about) the desired behavior of the object-level. The system must obey these assertions, although the assertions do not uniquely define the behavior of the system (i.e. systems are non-deterministic).

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 a technique called reflection principles. It shows to be 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.

4.4. Read-Only Reflective Representations

Another problem with the causal connection requirement is that often the complete downward causal connection of a self-representation cannot be realised. In many cases it is impossible to translate a modification to the reflective representation into the status and behavior of the system itself. For example, it is interesting to represent at the reflective level that

```
not(fact(X))
```

but how should this representation be enforced on the object-level? What if x is a fact at the object-level or what if x is later asserted?

Another argument against complete downward causal connection of the self-representation is that it is often not what the programmer of a reflective system wants. A programmer often wants to give systems a "read-only" reflective representation of certain aspects of themselves. Such representations make it possible for a system to reason about aspects of itself, without allowing it to act upon those aspects.

A possible solution to this problem is to introduce the notion of read-only reflective representations in the architecture. Consequently, the classification of the reflective representations contained in a self-representation as "read-only" reflective representations or as completely causally connected representations presents another design problem that has to be solved.[6]

5. The Problem of Reflective Overlap

When designing the meta-theory of a reflective architecture, one should be careful not to fall into the pit of **reflective overlap**. The problem of reflective overlap occurs when an aspect of a system that has been made explicit includes, or is included by, some aspect of the system that remains implicit (Smith,1986). It is the issue of reflective overlap that may cause the paradoxical situations that are sometimes associated with reflection.

For example, consider a reflective architecture which makes explicit the stack of the interpretation process running a system. The problem is that reflective computation also uses the run-time stack. So although the stack is made explicit, it at the same time includes an implicit aspect of the computation. It is therefore impossible to realise reflection in this case with only one representation of the run-time stack. Necessarily, there will have to be two representations: (i) the one that is actually used and (ii) the one that is explicitly represented and manipulated. Three representations of the stack will be necessary if we also want to make the stack of the reflective computation explicit.

The problem of reflective overlap introduces limits on the combination of choices that can be made in the design of the architecture. Whether the problem of reflective overlap will exist or not depends on the combination of meta-theory, level-interfacing mechanism and the causal connection technique.

The problem of reflective overlap is related to the undecidability results in meta-mathematics (Godel) (Russell). These results state that there are limits to how far a formal system can formalise aspects of itself. Reflective overlap states that there are limits to how far a computational system can have an explicit and modifiable self-representation of itself. An example which illustrates the problem is the well-known liar-paradox, of which the sentence

```
        false(this-sentence).            (i)
```

is a variant. This sentence is paradoxical, because its truth-value which is made explicit, also remains implicitly affected by the sentence itself.

In order to avoid reflective overlap it has to be checked whether in none of the states in which a system can get, reflective computation is performed which

(i) accesses an explicit representation X, and through that same computation

(ii) implicitly affects the aspect represented by X in such a way that the computation has become obsolete.

For example, the sentence (i) states a reflective fact about itself. But, this statement implicitly affects the sentence (i) (i.e. its truth-value) in such a way that the statement has become obsolete.

The problem of reflective overlap and its relation to meta-mathematics promises to be a fascinating future research topic.

6. A Classification of Existing Reflective Architectures

The previous sections discussed a three-dimensional design space formed by (i) the choice of meta-theories, (ii) the choice of means to program reflection and (iii) the choice of causal-connection mechanisms. If we situate the reflective architectures that have been built in this space only small areas seem to have been explored.

A first area is formed by reflective architectures in which the meta-theory makes explicit the operational aspect of systems. The terminology introduced by the meta-theory includes terms such as environments, interpreters, stacks, and continuations. Consequently, it does not provide a very interesting basis for reflective computation. Reflective computation is typically programmed explicitly in these architectures.

Computation in this class of architectures primarily happens at the object-level. The reflective level is more viewed as an extra facility, which is used in exceptional situations. The causal connection link of these architectures is realised by using the self-representation to actually implement the system. The larger part of the reflective architectures that have been built falls into this class. Examples are 3-LISP (Smith & Des Rivieres) and META-PROLOG (Bowen & Kowalski).

A second class of architectures that can be identified is concerned with the control aspect of systems. The meta-theory of these architectures makes explicit how deduction is controlled. Reflective computation is in these architectures activated implicitly. The reflective computation actually implements the control decisions that have to be made during interpretation of a system. This means that the causal connection requirement in these languages is again realised by an implementation relation. Examples of this class of architectures are SOAR and TEIRESIAS.

All of the reflective architectures that have been realised until now are procedural. A third class of reflective architectures, which is still under construction at the moment, attempts to realise declarative reflection. The meta-theory of these architectures (partially) makes explicit the semantic, and/or deductive aspects of systems. The reflective computation of a system is in these architectures implicitly programmed. Examples of attempts to build such architectures are MRS and GOLUX.

7. Conclusions

This chapter discussed issues in the design of a reflective architecture. It presented a three-dimensional space of designs that can be explored by reflective architectures. The emphasis of this discussion was on the relation between a particular design choice that is made and the possibilities for reflection this results in.

The first dimension of this space represents the meta-theories that can be adopted in a reflective architecture. The choice that has to be made here is extremely important because the reflective facilities of an architecture are theory-relative. The different aspects of a computational system the meta-theory can make explicit are: the syntactic aspect, the semantic aspect, the deductive aspect, the control aspect and the operational aspect. Meta-theories based on different combinations of those aspects result in entirely different reflective facilities.

The second dimension of the design space represents the different mechanisms that can be adopted for programming reflective computation. Again there is a range of designs imaginable here. Nevertheless the alternative designs were classified into those that support **implicit reflection**, those that support **explicit reflection**, and those that support both.

The third dimension of the design space represents the causal connection mechanism that can be adopted in a reflective architecture. On the basis of this dimension, designs for reflective architectures are classified into architectures for **procedural reflection**, in which the causal connection link is realised by an implementation relation, and architectures for **declarative reflection**, in which the causal connection link is realised by a specification relation. The former are easier to implement, but the latter result into more intresting reflective facilities.

Most reflective architectures that have been built fall into a few "cliche" areas of the design space. It would be an interesting future research topic to study in how far it is possible to build systems that deviate from those typical designs.

The next chapter illustrates the ideas presented in this chapter and the former one with a concrete example. It reports on an experiment that was performed in order to get a deeper understanding of the design and construction of reflective architectures.

NOTES

[1] (Smith,1987) defines a different three dimensional space that can be explored by reflective systems. Although this classification is based on different criteria, the outcome of the classification is the same: the existing architecture are also grouped in one small area of the three-dimensional space.

[2] Eventually, a reflective architecture could give computational systems multiple representations of themselves, suited for different reflective purpose. However, this would raise the

complexity of building and using a reflective architecture substantially.

[3] The "expressibility" of reflective sentences does not only depend on the meta-theory of the architecture. The meta-theory defines the names that can be used in reflective sentences. However, the syntax of the language defines the format these sentences may have.

[4] Des Rivieres (1986) discusses an example of such a space of reflective computations. He studies the power of a reflective architecture in which systems can talk about their program, environment and continuation.

[5] Note that architectures for declarative reflection might also present a solution to the problems of reflection discussed in chapter II, section 6. In this type of architecture more responsability over the control of reflective computation (and how it affects the object-computation) is given to the language-interpreter. Consequently this interpreter could be built such that it prevents negative impacts of the reflective computation on the overall computation.

[6] We do not follow B. Smith here, who demands that the causal connection between system and self-representation is maximal in both directions. This implies that (i) the self-representation should be able to engender the normal object-level. I.e. that the reflective deliberations can serve as one way of doing what is reflected about. And (ii) that the self-representation can always be translated in the behavior and status of the system itself. We believe that these strong requirements unneccessarily restrict reflective architectures to the least interesting class of architectures, namely architectures for procedural reflection.

CHAPTER VI

Implementing a Reflective Architecture

1. Introduction

This chapter reports on an experiment that was performed in building a reflective architecture. The purpose of the chapter is to illustrate the objects introduced in the previous chapters with a concrete example. The various steps that were taken in the implementation are treated at length.

Several valuable efforts to build reflective architectures in procedural languages, logic programming languages and production rule systems can be identified. Less work has been done on object-oriented languages (cf. section 7.2), although object-oriented languages are increasingly used as foundation for knowledge representation languages (cfr. KEE, LOOPS). It was therefore decided to do the experiment for an object-oriented language.

The experiment 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 modularity, encapsulation and abstraction facilities provided by object-oriented languages.

A second decision was made to do the experiment for an existing language. This ensures that the extra effort that has to be made to provide the language with a reflective architecture stands out clearly. The language KRS (Steels,1985) was chosen because (i) the know-how and implementation are available in our laboratory, (ii) KRS is designed not only as a programming language but also as a knowledge representation language, which opens a way to experiment with reflection for artificial intelligence programming (Maes,1986a) and (iii) KRS incorporates some advanced features such as lazy construction and consistency maintenance, which facilitate the construction of a reflective architecture (cf. next sections).

This chapter describes how the design and implementation of **KRS** were modified in order to produce a reflective variant of **KRS**, called **3-KRS**. However, the discussion is presented in a way that makes it transferable to any object-oriented language. At the end of the chapter the 3-KRS experiment is compared with existing object-oriented languages as well as with existing reflective architectures.

Chapter IV defined three steps that have to be taken when introducing a reflective architecture in a language. Before undertaking these steps, sentences in the language, such as messages or object-definitions, have to be data-structures of the language, so that it becomes possible to perform computation about them. This problem is solved in 3-KRS by introducing objects representing the syntactical constructs of the language and modifying the interpreter such that it translates the specific sentences it parses into instances of those objects. Once this is realised, a computational system consists of both data-objects and program-objects.

The first step is to build the self-representation of a computational system. A computational system in 3-KRS is given an object-oriented self-representation. Every object in the system has a one-to-one relation to a **meta-object**. The meta-object of an object X represents the explicit information about X (e.g. about its behavior and its implementation). The object X itself, on the other hand, groups the information about the entity of the domain it represents. Consequently, objects also serve as the unit of reflective computation.

The second step is the realisation of means to program reflective computation. An object in 3-KRS manipulates its meta-object in two cases. An object reflects in an implicit way whenever the interpreter does something with the object (e.g. the object is asked to handle a message). An object reflects in an explicit way whenever code in the object explicitly prescribes a manipulation of the meta-object.

The third step in the construction of a reflective architecture is the realisation of the causal connection link between an object and its meta-object. The solution adopted in 3-KRS is that a meta-object represents a meta-circular interpreter for its object: it implements the whole computation of its object. Meta-objects actually run the interpretation of the 3-KRS language in a meta-circular way. Consequently, a meta-object not only has the possibility to reason about its object, but it can actually act upon its object at run-time.

The next section briefly introduces the KRS language in order to allow a more technical account of these four steps. A more detailed description of the design of KRS can be found in (Steels,1986). (Van Marcke,1987) describes its implementation.

2. Brief Introduction to KRS

2.1. Data-Structures

A typical object-oriented language supplies building blocks, called **objects**, for constructing computational systems. An object has a set of **slots**. A slot relates the object to another object in the language or to an object in the underlying implementation language. This other object is called the **filler** of the slot. Sometimes the filler of a slot is indirectly defined. The **definition** of a slot is a LISP-form that has to be evaluated to find the filler.

A **snapshot** represents the internal structure of an object at a specific moment in time. Figure 1 presents snapshots of the objects Current-Year, Person and John.

```
<Current-Year> = <Number 1986>

<Person> =
    Type: <Object>
    Birth-Year: <Number-#3216>
    Age: {(>> (Minus (>> Birth-Year)) of Current-Year)}
    Name: <String-#1876>
    Younger-Than (?Another-Person): {(>> (Lessp (>> Age))
                                         Age of ?Another-Person)}

<John> =
    Type: <Person>
    Birth-Year: <Number 1961>
    Mother: <Mary>
    Father: <Object-#6789> =
                Type: <Person>
    Mean-Age-Of-Parents: {(// (+ (>> Age Mother)
                                 (>> Age Father))
                              2)}
    Best-Friend: {(>> Mother)}
```

Fig. 1. Snapshots of objects.

Objects are represented in a snapshot by triangular brackets. For example <Current-Year>, <Number 1986> and <Object-#6789> are all objects. An object may be followed by a description of the object. Such a description exists of "=" followed either by another object or by the slots of the object. The figure above contains descriptions of the objects <Current-Year>, <Person>, <John> and <Object-#6789>.

A slot is represented by its name, followed by a semicolon, followed by either the filler of the slot, or the definition of the slot (when enclosed by curly brackets). In the latter case, the definition of the slot has to be evaluated to find the filler. For example, the filler of the slot Best-friend for object John is the object that results from the evaluation of the LISP-form

```
(>> Mother)
```

Slots may have arguments. The slot Younger-Than, for example, takes as argument another person object. Arguments are preceeded by a question-mark. In order to find the filler of such a slot, the definition of the slot has to be evaluated with the arguments bound. For example, it is possible to find the filler of the Younger-Than slot for ?Another-Person bound to the object <George>.

Some objects in the language represent a LISP-structure. These objects have a print-name that has two parts. For example, <Number 1986> is an object of type Number whose contents is the LISP-number 1986. Similarly

```
<Form (+ 2 4)>
```

is an object of type Form and with contents the LISP-form (+ 2 4). These objects have slots that support primitive functions on the LISP structure, such as Minus, Plus, Eval, etc.

The objects of a computational system and the slot-links that exist between them define a directed graph, called the **object-graph**. The nodes of this graph represent objects, the arcs represents slots, and the labels on arcs represent the names of slots. Figure 2 presents such an object-graph. Note that the fillers that are not yet computed are not represented.

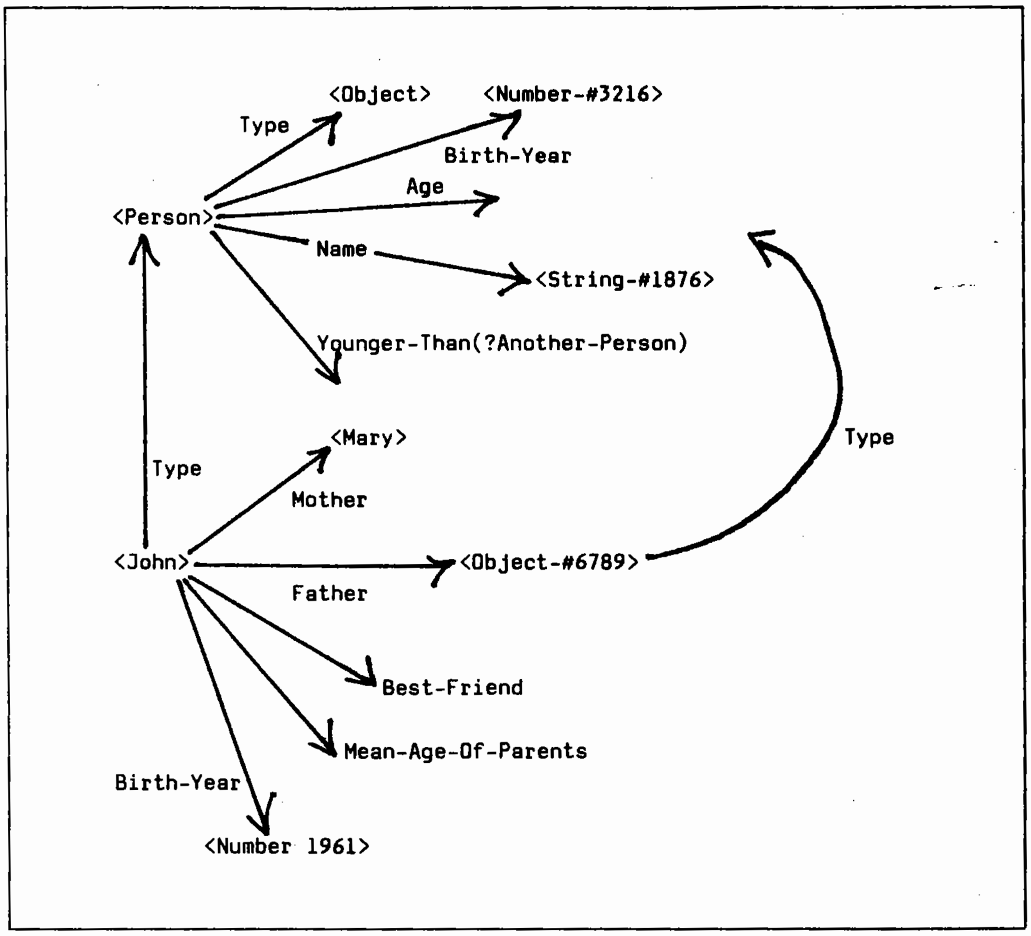


Fig. 2. The object-graph of a computational system.

2.2. Programs and Computation

2.2.1. Introduction

Computation consists of the exploration and manipulation of the object-graph. The **object-language** makes it possible to specify such computation. The object-language consists of functions for creating new objects and sending messages to objects[1]. Following a tradition in AI language design, these functions are implemented as LISP-functions. Computation happens when an object-language sentence is evaluated.

2.2.2. Sending a Message

By calling the function ">>", a message can be sent. The object the message is sent to, returns the filler of the slot asked in the message. It also possible to ask for a slot of the filler of a slot, and so on. Some examples, given the definitions of figure 1, where "->" means "returns",

```
(>> Type of Person) -> <Object>
(>> Mother of John) -> <Mary>
(>> Type Father of John) -> <Person>
(>> of Current-Year) -> <Current-Year>
(>> (Younger-Than (>> of George)) of John) -> <True>
(>> Best-Friend of John) -> <Mary>
```

The evaluation of the last two messages implicated the evaluation of other messages, namely the messages constituting the definition of the Younger-Than slot and of the Best-Friend slot

The object-language is lexically scoped. By default (i.e. when their is no "of <object>" part in the message) a message is sent to the object described by the outermost sentence in which the message is defined. We call this object the scope of the message. For example, the message

```
(>> Mother)
```

occuring in the sentence

```
(a Person
   (Best-Friend (>> Mother)))
```

will, when evaluated, be sent to the object that is created by the outermost sentence (in some languages this is called the "Self", i.e. (>> Mother) is like (>> Mother of Self)). Consequently, "(>>)" returns the object defined by the outermost sentence in which this empty message is defined ("return self" or (>> of Self)). Note that the snapshot in which a message occurs does not necessarily represent the scope of the message. If this is the case, we will make the scope of the message explicit in the snapshot.

2.2.3. Creating an Object

The functions "a/an" and "defobject" create a new anonymous object and a new named object respectively. For example

```
(a Person) -> <Person-#6789>
(defobject John (a Person)) -> <John>
(a Person
   (Birth-Year (a Number))
   (Younger-Than (?Another-Person)
       (>> (Lessp (>> Age)) Age of ?Another-Person))) -> <Person-#2876>
```

Staight brackets are used to create objects representing LISP structures. For example

```
[Symbol Age] -> <Symbol Age>
[Number 10] -> <Number 10>
[List (3 5)] -> <List (3 5)>
[Form (+ 3 9)] -> <Form (+ 3 9)>
```

Note that this language makes no distinction between "class objects" and "terminal-instance objects" (in contrast with languages such as SMALLTALK and LOOPS). Every object can serve as the "prototype" for the creation of a new object. For example, the object John could fill the Type slot of the object George.

2.3. Inheritance

Objects inherit information from their type. KRS incorporates a single inheritance hierarchy with at the top the object with name Object. An object inherits the slots it does not redefine by (lexically) copying these slots from its type. For example, the object John in figure 1 will answer messages as if it was defined as

```
<John> =
    Type: <Person>
    Birth-Year: <Number 1961>
    Mother: <Mary>
    Father: <Object-#6789> =
                Type: <Person>
    Best-Friend: {(>> Mother)}
    Mean-Age-Difference-With-Parents: {(// (+ (>> Age Mother)
                                              (>> Age Father))
                                           2)}
    Age: {(>> (Minus (>> Birth-Year)) of Current-Year)}
    Name: <String-#1876>
    Younger-Than (?Another-Person): {(>> (Lessp (>> Age))
                                         Age of ?Another-Person)}
```

The slots in italics are (virtually) copied from the object Person. Consequently

```
(>> Name of John) -> <String-#1876>>
(>> Age of John) -> <Number 25>
```

Note that the former filler only partially defines the name of object John. The name of John is a string, however the contents of this string is not known yet.

3. Representing Programs as Objects

In order to make it possible to consider programs as data of reflective computation, programs have to be data-structures of the language. This problem is solved in 3-KRS by representing programs as objects. Whenever the 3-KRS interpreter parses an object-language sentence or a LISP-form, it turns it into an object. For example, when at some point the sentence

```
(>> Mother of John)
```

is used, the 3-KRS interpreter creates an object to represent it (Message-#2351 in figure 3).

```
<Message-#2351> =
   Type: <Message>
   Slot-List: <List (Mother)>
   Target: <John>
   Eval: {<Form (krs:process-message (>> Slot-List)
                                      (>> Target))>}
```

Fig. 3. Programs are objects in 3-KRS.

We call such an object a **program-object**. The object Message-#2351 has slots representing the Type, Slot-List, and Target of the message. It also has a slot with name Eval which represents the result of evaluating the message. Note that the forms that do not belong to the object-language, e.g. the LISP-form

```
(krs:process-message (>> Slot-List)
                     (>> Target))
```

are turned into objects of type Form.

From this moment on, all definitions of slots will be turned into objects before being evaluated (because definitions are object-language sentences or LISP-forms). Consequently, we have to modify the rule for getting the filler of a slot by means of its definition. The new rule is that the filler of a slot is equal to the result of sending the message Eval to the object representing the definition. For example, asking for the filler of the Eval slot of Message-#2351 will result in sending the message Eval to the object

```
<Form (krs:process-message (>> Slot-List)
                           (>> Target))>
```

When an object representing a LISP structure gets an Eval message, it returns the evaluation of the LISP structure that it contains. So in the example, the form

```
(krs:process-message (>> Slot-List)
                     (>> Target))
```

is evaluated. Krs:process-message is a function of the underlying implementation of 3-KRS, which is used to process a message. It returns

```
<Mary>
```

The mechanism for making objects of programs is implemented by introducing an object for every function of the object-language. This set includes the objects Message, Object-Definition, and Instance-Creation shown below.

```
<Message> =
   Type: <Program-Object>
   Slot-List: <List-#2987>
   Target: <Object-#7658>
   Eval: {<Form (krs:process-message (>> Slot-List) (>> Target))>}

<Object-Definition> =
   Type: <Program-Object>
   New-Object-Name: <Symbol-#6545>
   Slot-Description-List: <List-#3568>
   Eval: {<Form (krs:create-object-with-Name (>> New-Object-Name)
                                             (>> Slot-Description-List))>}

<Instance-Creation> =
   Type: <Program-Object>
   New-Instance-Type: <Object-#8787>
   Slot-Description-List: <List-#8768>
   Eval: {<Form (krs:create-instance-of (>> New-Instance-Type)
                                        (>> Slot-Description-List))>}
```

Fig. 4. The syntactical constructs of the object-language are represented as objects.

Ad read-time, an extra parsing step translates every object-language sentence into a program-object of one of these types. Consequently, the object John presented in figure 1, will now look as follows

```
<John> =
    Type: <Person>
    Birth-Year: <Number 1961>
    Mother: <Mary>
    Father: <Object-#6789> =
                Type: <Person>
    Mean-Age-Difference-With-Parents: {<Form (// (+ (>> Age Mother)
                                                     (>> Age Father))
                                                  2))>}
    Best-Friend: {<Message-#7878> =
                     Type: <Message>
                     Slot-List: <List (Mother)>
                     Target: <John>
                     Eval: {<Form (krs:process-message
                                        (>> Slot-List Best-Friend)   (i)
                                        (>> Target Best-Friend))>}>}
```

Note that from now on a snapshot (representing the internal structure of an object) looks different. All sentences that occur in the description of an object have internally become represented as program-objects. In order to make the new snapshots more compact, the notation

```
        <Message (>> Mother of John)>
```

is used as a shorthand for the message-object with target the object John and the Slot-List the list (Mother). Similar shorthands are also used for instances of the objects Instance-Creation and Object-Definition.

However, if all programs would be transformed into objects this would cause an infinite behavior. The creation of a message-object would for example cause an infinite loop, because the Eval slot of a message would always involve the creation of other message-objects. To avoid this, forms within the contents of a Form object are no longer expanded into objects. For example, the two messages in (i) are not turned into message-objects before being evaluated.

When the object John is asked for the filler of his Best-Friend slot, as in

```
        (>> Best-Friend of John)
```

this results in the sending of the message

```
        (>> Eval)
```

to the object <Message-#7878> (because of the definition-filler rule), which results in the sending of the message

```
        (>> Eval)
```

to the object

```
        <Form (krs:process-message (>> Slot-List Best-Friend)
                                   (>> Target Best-Friend))>      (i)
```

which returns the evaluation of the embedded LISP form, which is

```
        <Mary>
```

We are now ready to consider the various issues involved in building a reflective architecture.

4. The Self-Representation of an Object-Oriented System

4.1. Introduction

The first problem to be handled when building a reflective architecture was called the problem of the self-representation of a computational system (cf. chapter IV). The interpreter of the language has to be able to build an explicit representation of any computational system it is running. A computational system in an object-oriented language consists of an object-graph of objects (program-objects and data-objects) that send messages to each other. 3-KRS gives such a computational system an object-oriented self-representation.

Every object in the object-graph has a slot with name **Meta** which links the object to a representation of itself, called the **meta-object** of the object. There is a one-to-one relationship between objects and meta-objects. A meta-object has a slot with name **Referent** which is again filled by the object it is a representation of.

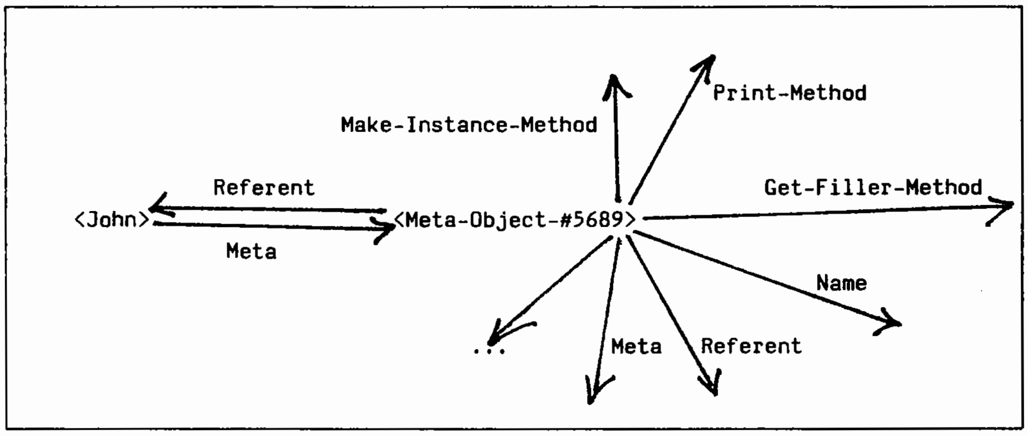


Fig. 5. An object and its meta-object.

The meta-object of an object groups all the information about the object that is made explicit. It has information about its syntactical form, its semantics, its computation and its implementation. It contains for example the Methods to make an instance of the object, to print the object and to let the object handle a message. It also stores information such as the name of the object, documentation about the object, etc.

Every object in the object-graph has a meta-object. Thus also meta-objects and meta-objects of meta-objects, and so on. This unbounded list of meta-objects is only implementable if meta-objects are created in a lazy way, i.e. only when they are actually needed (when they have to answer a message). The 3-KRS interpreter incorporates such a lazy-construction mechanism (Van Marcke,1987).

4.2. The Meta-Theory

4.2.1. Introduction

The meta-theory of an object-oriented reflective architecture is represented in the object-graph. Every object-oriented language has a set of **primitive objects**. The primitive objects of a language are those objects that are created by the interpreter of an object-oriented language when a session with the language is started. The set of primitive objects for example includes the objects Object, Slot, Number, etc. In an object-oriented language where programs are also objects, the set of primitive objects also includes the program-objects, such as Message, Object-Definition and Instance-Creation.

The meta-theory of an object-oriented language is represented in the meta-objects of those primitive objects. Any object that is created is necessarily a specialisation of one of those primitive objects. Consequently, the contents of the meta-object of the new object is automatically determined through inheritance (unless the object overrides the slot Meta, as discussed in section 6).

$$\begin{tikzcd}
 & \text{<Object>} \arrow[rrr, shift left, "\text{Meta}"] & & & \text{<Meta-Object>} \arrow[lll, shift left, "\text{Referent}"] \\
\text{<Number>} \arrow[ur, "\text{Type}"] \arrow[d, "\text{Meta}"] & \dots \arrow[u, "\text{Type}"] & & \text{<Program-Object>} \arrow[ull, "\text{Type}"] & \\
\text{<Number-Meta>} & & \text{Message} \arrow[ur, "\text{Type}"] \arrow[d, "\text{Meta}"] & \text{Object-Definition} \arrow[u, "\text{Type}"] \arrow[d, "\text{Meta}"] & \dots \arrow[ul, "\text{Type}"] \\
 & & \text{<Message-Meta>} & \text{<Object-Definition-Meta>} &
\end{tikzcd}$$

Fig. 5. The meta-theory of 3-KRS is embodied in the primitive objects of the language.

We call the meta-objects of the primitive objects the **primitive meta-objects** of the reflective architecture.

4.2.2. The Default Meta-Object

The object with name "Object" is the topnode of the inheritance-hierarchy. Any object in the object-graph is a specialisation of the object Object. Consequently, the meta-object of the object Object is the default meta-object that is created for an object. The meta of Object is the object with name Meta-Object.

```
<Object> =
   Meta: <Meta-Object> =
            Referent: {<Message (>>)>}
```

Fig. 6. The most general object with its meta-object.

Meta-Object represents the meta-theory the language incorporates about objects in general. It includes all the information about objects the reflective architecture makes explicit. Chapter V has analysed a language in terms of a syntactical aspect, a denotational aspect, a computational/deductional aspect and an operational aspect. The Meta-Object of an object-oriented language may formalise these four aspects of objects in the language.

As an example, figure 7 illustrates the structure of the object Meta-Object of 3-KRS. Meta-Object-#4568 is the instance of the default meta-object that is created for the object George.

```
<George> =
   Type: <Person>
   Birth-Year: <Number 1955>
   Meta: <Meta-Object-#4568>

<Meta-Object-#4568> =
   Type: <Meta-Object>
   Referent: <George>
   Name: {<Instance-Creation
            (a Symbol
               (Contents [Form (krs:find-object-name
                                   (>> Referent))]))>}              (i)
   Slots: {<Instance-Creation
            (a List
               (Contents [Form (krs:get-slots-of
                                   (>> Referent))]))>}              (i)
   Get-Filler-Method (?Slot-Name):
            <Form (krs:get-filler-of (>> Referent)
                                   ?Slot-Name)>                     (ii)
   Inherit-Slot-Method (?Slot-Name):
           <Form (krs:inherit-slot (>> Referent)
                                 ?Slot-Name)>                       (ii)
   Add-Slots-Method (?Slot-Description-List):
           <Form (krs:make-slots (>> Referent)
                               ?Slot-description-List)>            (iii)
   Make-Instance-Method: <Form (krs:make-instance (>> Referent))>  (iii)
   Print-Method: <Form (krs:object-print-self (>> Referent))>      (iii)
```

Fig. 7. The default meta-object of a 3-KRS object.

A Meta-Object holds information about the syntactical aspect of the object it is about (called its referent hereafter) (see (i)). It has a slot representing the name of its referent (or the identification-number of the referent for anonymous objects), and a slot representing the list of slot-names of its referent. So, for the above example

```
(>> Name Meta of George) -> <Symbol George>
(>> Slots Meta of George) -> <List (Birth-Year Type Meta)>
```

Note that getting the filler of these slots involves descending in the LISP implementation of the 3-KRS language. So these slots actually represent reifications of structures in the underlying LISP implementation. For example, the function

```
krs:find-object-name
```

is a function that retrieves the association for the property "Name" in the LISP association-list that internally represents an object.

The 3-KRS Meta-Object incorporates explicit information about the computational aspect of an object (see (ii)). An object does computation when it is asked for the filler of a slot. The Meta-Object represents the computation of an object. The slots with name Inherit-Slot-Method and Get-Filler-Method are filled with the procedures the object uses to inherit a slot and to get the filler of a slot respectively. These procedures are again a reification of the procedures used by the underlying LISP implementation for these tasks. For the above example

```
(>> (Get-Filler-Method [Symbol Birth-Year]) Meta of George)
         -> <Form (krs:get-filler-of
                       (>> Referent of Meta-Object-#4568)
                       [Symbol Birth-Year])>
(>> Eval (Get-Filler-Method [Symbol Birth-Year]) Meta of George)
         -> <Number 1955>
(>> (Inherit-Slot-Method [Symbol Age]) Meta of George)
         -> <Form (krs:inherit-slot
                       (>> Referent of Meta-Object-#4568)
                       [Symbol Age])>
(>> Eval (Inherit-Slot-Method [Symbol Age]) Meta of George)
         -> <Number 25>
```

Finally the 3-KRS Meta-Object represents the operational aspect of its referent (see (iii)). It incorporates explicit representations of the things the interpreter does with an object. The Meta-Object has slots that represent the procedures to add slots to the referent, to make an instance of the referent, to print the referent, etc.

The operational slots were chosen on the basis of an in-depth analysis of the LISP implementation of 3-KRS. The 3-KRS interpreter was reorganised so that its structure reflected the basic actions an interpreter does with an object. Afterwards these actions were reified in the slots of the Meta-Object. For the above example

```
(>> (Add-Slots-Method [List ((Birth-Year [Number 1966])
                             (Brother (>> of John)))]) Meta of George)
      -> <Form (krs:make-slots (>> Referent of Meta-Object-#4568)
                               [List ((Birth-Year [Number 1966])
                                      (Brother (>> of John)))])>
(>> Eval (Add-Slots-Method [List ((Birth-Year [Number 1966])
                                  (Brother (>> of John)))])
      Meta of George)
      -> <George> =
            Type: <Person>
            Birth-Year: <Number 1966>
            Meta: <Meta-Object-#4568>
            Brother: <John>
(>> Make-Instance-Method Meta of George)
      -> <Form (krs:make-instance
                  (>> Referent of Meta-Object-#4568))>
(>> Eval Make-Instance-Method Meta of George)
      -> <George-#7890>
(>> Print-Method Meta of George)
      -> <Form (krs:object-print-self
                  (>> Referent of Meta-Object-#4568))>
(>> Eval Print-Method Meta of George)
      -> <George>
```

4.2.3. The Meta-Objects of Program-Objects

The Meta-Object object represents the information about an object that is made explicit in an object-oriented reflective architecture. About certain types of objects, more interpreter-information can be made explicit. In particular, the language-interpreter embodies more information about the denotational aspect of program-objects. It incorporates information about how the evaluation of a program-object should be computed. This information may be made explicit in the meta-object of program-objects.

The 3-KRS interpreter, for example, attributes more specialised meta-objects to program-objects. The meta-object of a program-object has one extra slot with name "Eval-Method". This slot represents the procedure that is used for evaluating the program-object. The filler of this slot can be used to compute the evaluation of the program-object. As an example, figure 8 presents the meta-object that is constructed for message-objects.

```
<Message> =
   Type: <Program-Object>
   Slot-List: <List-#2987>
   Target: <Object-#7658>
   Meta: <Message-Meta>
   Eval: {<Form (krs:process-message (>> Slot-List)
                                     (>> Target))>}

<Message-Meta> =
   Referent: <Message>
   Type: <Meta-Object>
   Eval-Method:
      <If-#6789> =
         Condition: <Message (>> Emptyp Slot-List Referent)>
         Then-Part: <Message (>> Target Referent)>
         Else-Part: <Message-#6704> =
                       Type: <Message>
                       Target:
                          {<Form (krs:get-filler-of
                                     (>> Target Referent)
                                     (>> Last Slot-List Referent))>}
                       Slot-List: {<Message
                                      (>> But-Last Slot-List Referent)>}
```

Fig. 8. The meta-object of a message-object.

The meta-object of the object Message is the object Message-Meta. The Message-Meta object inherits from the Meta-Object object and adds a slot with name Eval-Method. This slot is filled with a program to compute the evaluation of a message. When this program is evaluated, it returns the target of the message if the slot-list of the message is empty. For example, it will return the object George if the referent is the message

```
(>> of George)
```

If the slot-list of the message is a non-empty list ($name_1$.. $name_n$), it asks the target of the message for the filler of $name_n$ and sends the resulting object the message with slot-list ($name_1$ .. $name_{n-1}$). For example, it will return the result of the message

```
(>> Type)
```

sent to <Number 1955>, when the referent is the message

```
(>> Type Birth-Year of George>
```

When a particular message-object is created by the interpreter, for example

```
<Message-#2351> =
   Type: <Message>
   Meta-Object: <Message-Meta-#6745> =
                 Referent: <Message-#2351>
                 Type: <Message-Meta>
   Path: <List (Birth-Year)>
   Target: <George>
```

this message-object by inheritance receives a Message-Meta object as its meta-object. This meta-object inherits the Eval-Method slot from the Message-Meta object. Consequently

```
(>> Eval-Method Meta of Message-#2351) -> <If-#2312>
(>> Eval Eval-Method Meta of Message-#2351) -> <Number 1955>
```

which is equal to the result returned by

```
(>> Eval of Message-#2351)
```

Similarly specialised meta-objects were defined for the other program-objects of 3-KRS. Together these meta-objects embody the meta-theory 3-KRS makes explicit about programs.

4.3. The Reference Mechanism

The self-representation mechanism discussed above adopts a very straightforward reference mechanism. The reflective representations that are relevant to a specific object are grouped in one meta-object, and there exists a bidirectional link between the object and its meta-object. The referent of a reflective representation is explicitly mentioned in the reflective representation and the reflective representation of an object are explicitly mentioned in that object.

However, because of inheritance, it is not necessary to specify for every object what meta-object has to be created. The inheritance mechanism of an object-oriented language makes it possible to create reflective representations that are relevant to a whole set of objects, instead of just one. In particular, a specific meta-object is relevant to all the specialisations of its referent, unless these specialisations override the slot with name Meta.

Note that this reference mechanism is more elegant than the reference mechanisms of the languages discussed in chapter IV and chapter V. Because these reference mechanisms are rather weak, the meta-theory of those languages consists of one program representing the meta-circular interpreter. For example, in 3-LISP the complete meta-theory of the language is represented in one meta-circular version of the evaluator. The link between a particular syntactical construct and its denotational aspect is not made explicit. Consequently, every sentence in the language receives the same self-representation, being this meta-circular interpreter. Although this self-representation makes explicit the operational aspect of the sentence (what it returns after evaluation), it does not make explicit the denotational aspect of the sentence (how its evaluation is defined).

The meta-theory of 3-KRS is distributed over the objects that represent the constructs of the language. 3-KRS makes explicit what part of the meta-circular interpreter represents the interpretation of a particular syntactical construct. In addition, 3-KRS makes explicit the denotational aspect of a particular syntactical construct of 3-KRS. This means that in 3-KRS systems are able to do reflective computation about the denotational aspect of a particular sentence as well as about its operational aspect.

5. The Causal Connection Requirement

5.1. Introduction

Before discussing the problem of programming object-oriented reflective computation, we first discuss the third problem that has to be solved when building a reflective architecture. This problem states that there should be a causal connection link between a computational system and its self-representation. Consequently meta-objects and their objects should be related to each other such that a change to one, is reflected in the other. 3-KRS has an architecture for procedural reflection. The causal connection problem is solved by a meta-circular interpreter (cf chapter IV).

For an object-oriented self-representation, this means that the meta-object of an object should actually be used to implement the object. Consequently, whenever something is changed in the meta-object, the behavior of the object itself will automatically comply with these modifications. Vice versa every change (from the outside) to the object will necessarily happen through the meta-object, so that the meta-object always contains an updated representation of the object. So the meta-circular interpreter guarantees the causal connection between an object and its meta-object in both directions.

The meta-circularity of the 3-KRS interpreter is realised in 3-KRS by making a number of small modifications to the primitive program-objects and to the LISP-implementation of the language.

5.2. Meta-Circularity of Program-Objects

Section 4.2.3. discussed how an explicit representation of the denotational aspect of a syntactical construct is embedded in the meta-object of the program-object that represents this syntactical construct. Some modifications have to be made to ensure that this representation is also actually used by the system when evaluating the program-object. Figure 9 shows the modifications that were necessary:

```
<Program-Object> =                ;;Previous Definition
   Type: <Object>
   Meta: <Meta-Object-#4376>

<Program-Object> =                ;;Modified Definition
   Type: <Object>
   Meta: <Meta-Object-#4376>
   Eval: {<Form (>> Eval Eval-Method Meta)>}

<Message> =                       ;;Previous Definition
   Type: <Program-Object>
   Slot-List: <List-#2987>
   Target: <Object-#7658>
   Meta: <Message-Meta>
   Eval: {<Form (krs:process-message (>> Slot-List)
                                     (>> Target))>}

<Message> =                       ;;Modified Definition
   Type: <Program-Object>
   Slot-List: <List-#2987>
   Target: <Object-#7658>
   Meta: <Message-Meta>
```

Fig. 9. Making the interpreter meta-circular - part 1.

The evaluation of program-objects will now happen in a meta-circular way. For example, when the object <Message-#8787> is asked for the filler of its Eval slot, as in

```
(>> Eval of Message-#8787)
```

this slot will be inherited from the object Program-Object. Consequently, the evaluation of the message is reduced to the sending of the message Eval to

```
<Form (>> Eval Eval-Method Meta of Message-#8787)>
```

which results in the evaluation of

```
(>> Eval Eval-Method Meta of Message-#8787)
```

The evaluation of this message happens by the LISP implementation because a Form-object returns the LISP-evaluation of the form that it contains when an Eval message is sent to it. The Message-Meta object (as defined earlier) looks as follows

```
<Message-Meta> =
   Referent: <Message>
   Type: <Meta-Object>
   Eval-Method:
      <If-#6789> =
         Condition: <Message (>> Emptyp Slot-List Referent)>       *
         Then-Part: <Message (>> Target Referent)>                 *
         Else-Part: <Message-#6704> =                               *
                       Type: <Message>
                       Target:
                          {<Form (krs:get-filler-of
                                     (>> Target Referent)
                                     (>> Last Slot-List Referent))>}
                       Slot-List: {<Message
                                        (>> But-Last Slot-List Referent)>}
```

The Eval-Method will be equal to the object If-#6789.

So the evaluation of a message-object is handled explicitly by its meta-object. Note however that sending the message Eval to the Eval-Method slot of the Message-Meta object involves the evaluation of new message-objects (see *). A second (implicit) evaluation scheme is needed in order to guarantee the finiteness of this explicit scheme.

To avoid infinite regress the Eval-Method of the primitive meta-objects is emulated by the LISP-implementation of 3-KRS. The Eval-Method slot of the meta-object of a message is only used when it deviates from the default. So actually the Program-Object is defined as in figure 10. If the program-object has a special meta-object, the evaluation of the program-object is delegated to this meta-object. If the program-object has a copy of one of the primitive meta-objects, its evaluation is handled by a function of the LISP-implementation. This function exactly causes the same behavior as the Eval-Method of a primitive meta-object.

```
<Program-Object> =
   Type: <Object>
   Meta: <Meta-Object-#4376>
   Eval: {<Form (if (special-meta-object-p (>>))
                       (>> Eval Eval-Method Meta)
                       (krs:process-sentence (>>))))>}
```

Fig. 10. Making the interpreter meta-circular - part 1 revisited.

5.3. Meta-Circularity of Objects

The previous section discussed the modifications that have to be made in order to guarantee the causal connection of the Eval-Method of the Meta of a program-object. Similar modifications have to be made in order to guarantee the causal connection of the slots defined by Meta-Object (cf. figure 7). The interpretation of an object should involve using the slots of its meta-object instead of the implicit implementation. As an example, figure 11 highlights the change this implies upon the LISP-implementation of the 3-KRS language for the Get-Filler-Method. The function krs:get-filler-of is used in the LISP-implementation to compute the filler of a slot.

```
(defun krs:get-filler-of (object slot-name)
      (eval '(>> Eval (Get-Filler-Method [Symbol ,slot-name])
                    Meta of ,(find-object-name object)))
```

Fig. 11. Making the interpreter meta-circular - part 2.

This modification ensures that the meta-objects of 3-KRS objects are also actually used during the interpretation of the language. Consequently, the interpretation also becomes object-oriented. Whenever something has to happen with an object, this will be asked to the meta-object of the object. For example, whenever a message

```
(>> Birth-Year of George)
```

has to be evaluated, this will result in the evaluation of

```
(>> Eval (Get-Filler-Method [Symbol Birth-Year]) Meta of George)
```

This example illustrated how the explicit representation of the behavior and implementation of objects is causally connected to their actual behavior and implementation by means of an object-oriented, meta-circular interpreter. However, we again need a second interpreter to make this interpretation scheme actually work. The above example clearly indicates the infinite behavior of the meta-circular interpreter. In order to get the filler of a slot for an object, we have to ask the meta-object of this object for the filler of the Get-Filler-Method slot, consequently we have to ask the meta-object of the meta-object how to get the filler of a slot of the meta-object, and so on.

The technique that is used to avoid this infinite regress is the same as the one used for program-objects: the slots of primitive meta-objects are emulated by the LISP-implementation of 3-KRS. When an object has (a copy of) a primitive meta-object, the object will be handled by the LISP implementation of 3-KRS: the LISP-implementation will compute how the object responds to a message, how an instance of the object is created, how the object is evaluated, etc. The meta-object of an object is only used to interpret the object when this meta-object deviates from the basic meta-objects, e.g. when the meta-object specifies another form of inheritance or another way to create instances for the object. So actually the modifications made to the LISP-implementation look as in figure 12.

```
(defun krs:get-filler-of (object slot-name)
   (if (special-meta-object-p object)
       (eval '(>> Eval (Get-Filler-Method [Symbol ,slot-name])
                   Meta of ,(find-object-name object)))
       (krs:default-get-filler-of object slot-name)))

(defun krs:default-get-filler-of (object slot-name)
   ... ;;the LISP-implementation of krs:get-filler-of
   )
```

Fig. 12. Making the Interpreter meta-circular - part 2 revisited.

Some conditionals were added which take care that, if the object has a special meta-object, the interpretation of the object is explicitly handled instead of implicitly. Similar modifications to the LISP-implementation have to be made for the other slots of the Meta-Object object. Consequently, the interpretation of the object created by

```
(defobject George
   (a Person
      (Birth-Year [Number 1977])))
```

is handled by the LISP implementation of 3-KRS. On the other hand, the interpretation of the object created by

```
(defobject Bill
   (a Person
      (Meta (a Meta-Object
               (Get-Filler-Method (?Slot-Name)
                    [Form (>> (>> Contents of ?Slot-Name) of George)])))))
```

is handled by its meta-object.

To summarise how the meta-circular interpretation of 3-KRS works, consider the message

```
(>> Birth-Year of Bill)
```

The moment this message is parsed, the interpreter turns it into an object

```
<Message-#7890> =
   Type: <Message>
   Meta: <Message-Meta-#3454>
   Target: <Bill>
   Slot-List: <List (Birth-Year)>
```

When the message has to be evaluated at some point, the following computation will take place, where "=>*" stands for "reduces-after-some-steps-to":

```
(>> Eval) sent to  <Message-#7890>

      =>*  ;;because this Message inherits the Eval slot from
           ;;the object Program-Object we get:
(>> Eval) sent to <Form (if (special-meta-object-p (>>))
                            (>> Eval Eval-Method Meta)
                            (krs:process-sentence (>>)))>
           ;;in which the scope is <Message-#7890>,
      =>*  ;;because sending Eval to a Form returns the
           ;;LISP evaluation of the embedded form, this results in the
           ;;evaluation of:
(if (special-meta-object-p (>>))
    (>> Eval Eval-Method Meta)
    (krs:process-sentence (>>)))
           ;;in which the scope is <Message-#7890>

      =>*  ;;because the meta of <Message-#7890> is a copy of the
           ;;Meta-Object object, i.e not special, we get the evaluation of:
(krs:process-sentence (>>))
           ;;in which the scope is <Message-#7890>

      =>*  ;;because the evaluation of this function-call results
           ;;in a call of krs:get-filler, we get the evaluation of
(krs:get-filler-of object slot-name)
           ;;with object bound to <Bill> and slot-name bound to Birth-Year

           ;;because of the definition of the function krs:get-filler, we get:
(if (special-meta-object-p object)
    (eval '(>> Eval (Get-Filler-Method [Symbol ,slot-name])
               Meta of ,(find-object-name object)))
    (krs:default-get-filler-of object slot-name))
           ;;with object bound to <Bill> and slot-name bound to Birth-Year

      =>*  ;;because the meta-object of <Bill> is <Meta-Object-#2314>,
           ;;which is special, we get the evaluation of:
(eval '(>> Eval (Get-Filler-Method [Symbol ,slot-name])
           Meta of ,(find-object-name object)))
           ;;with object bound to <Bill> and slot-name bound to Birth-Year

      =>*  ;;this results in the evaluation of the message
(>> Eval (Get-Filler-Method [Symbol Birth-Year]) Meta of Bill)

      =>*  ;;because the meta-object of <Bill> is <Meta-Object-#2314>,
           ;;we get:
(>> Eval (Get-Filler-Method [Symbol Birth-Year])) sent to <Meta-Object-#2314>

      =>*  ;;because of the definition of the Get-Filler-Method
           ;;of <Meta-Object-#2314>, we get:
(>> Eval) sent to <Form (>> (>> Contents of [Symbol Birth-Year]) of George)>

      =>*  ;;because the evaluation of a Form object is the
           ;;evaluation of the LISP form that it contains we get the
           ;;evaluation of:
(>> (>> Contents of [Symbol Birth-Year]) of George)

      =>*  ;;which results in the evaluation of
(>> Birth-Year of George)

      =>*  ;;the same process starts over for the object George
                        ;;instead of the object Bill, but because George has
                        ;;a default-meta-object, this results in the
                        ;;evaluation of
```

```
(krs:default-get-filler-of <George> <Symbol Birth-Year>)

      =>*  ;;which represents the way the LISP-implementation gets the
           ;;filler of a slot for an object, it returns:
<Number 1955>
```

Note finally that the language has to incorporate a form of **consistency maintenance** in order to make the causal connection mechanism discussed above work. More specifically, the interpreter has to take care that references in the language always refer to the latest version of an object. This overcomes the situation in which a meta-object redefines its object during reflection, and returns to the inconsistent version of the object (the old version which triggered the reflective computation) afterwards. KRS has such a form of consistency maintenance (Van Marcke,1986).

6. Programming Object-Oriented Reflection

6.1. Introduction

The second problem in the construction of a reflective architecture was called the problem of programming reflective computation. A reflective architecture has to provide conditions in which an object switches from object-computation to reflective computation and vice versa. 3-KRS has an architecture which supports implicit and explicit reflective computation (cf. chapter V).

An important advantage of 3-KRS over already built languages is that the reflective computation of an object may be coupled to the flow of the object-computation (as in 3-LISP), or uncoupled from it (as in SOAR). An object may prescribe implicit (uncoupled) reflective computation by means of a meta-object that deviates from the primitive meta-objects. When an object has a deviating meta-object, its interpretation will be handled by this meta-object instead of the LISP implementation of the language. On the other hand, an object may also explicitly prescribe (coupled) reflective computation. The code in an object may contain pieces of reflective code, i.e. sentences that prescribe the manipulation of a meta-object.

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 prototype person object reflective. Or one can make (almost) all messages in the system behave in a special way, by making the prototype message object reflective. Note that in a language like 3-LISP this is not the case. 3-LISP for example does not provide the possibility to specify reflective behavior for a class of expressions.

6.2. Implicit Reflection

6.2.1. Local Implicit Reflection

Every object in a computational system has the possibility to perform implicit reflective computation. The computation of an object will make a level-shift whenever the interpreter has to perform an action on the object and the object has a meta-object deviating from the primitive meta-objects.

The slots of a deviating meta-object prescribe in an explicit way how the interpreter action should be performed instead, i.e. what reflective computation should be performed at that point in the interpretation. So, for each slot in the meta-object that implements a part of the interpreter, it is possible to program reflective computation for the object. The slots of the meta-object are the hooks in the interpreter where reflective computation can be triggered.

Most of the time a special meta-object is created as a specialisation of an existing meta-object, as illustrated in figure 13.

```
<The-Current-Date> = <List-#3423> =
                        Contents: {<Form (get-time)>}

<The-Current-User> = <Symbol-#8909> =
                        Contents: {<Form (user-id)>}

<Annotating-Meta> =
    Type: <Meta-Object> =
            Make-Instance-Method:
               <Form
                 (let ((?the-instance
                          (>> Eval Default-Make-Instance-Method))
                       (?the-current-date (>> of The-Current-Date))
                       (?the-current-user (>> of The-Current-User)))
                    (>> Eval
                        (Add-Slots-Method
                            [List ((When-Created ?the-current-date)
                                   (Author ?the-current-user))])
                        Meta Meta of ?the-instance)
                    (>> Eval
                        (Add-Slots-Method
                            [List ((Most-Recently-Created
                                    ?the-instance))]))
                    ?the-instance)>
```

Fig. 13. A deviating meta-object - 1.

Annotating-Meta inherits from the Meta-Object object. It overrides the Make-Instance slot such that the instances of its referent are documented at the time of their creation. Actually it creates an instance using the default method to create instances, and then performs some other actions. Since this procedure is often used when defining meta-objects, every meta-object in the 3-KRS system also stores the default interpretation methods in the slots with names Default-Add-Slots-Method, Default-Make-Instance-Method, and so on. When we now define

```
(defobject Dossier
   (Meta (an Annotating-Meta
             (Referent (>>)))))
```

and we create an instance of Dossier, as in

```
(defobject My-Dossier
   (a Dossier))
```

the filler of the slot Make-Instance-Method of the meta-object of Dossier will be evaluated, since the meta-object of Dossier deviates from the primitive meta-objects. In the Make-Instance-Method, the Dossier object itself (because a meta-object has a link to its object) and an explicit representation of its behavior and implementation (in terms of the other slots of the meta-object) are accessible to reason about or to make modifications. E.g. the object can be questioned about its current state, this state can be modified, extra slots for the object can be created, or its future behavior can be modified.

The meta-object in the example above modifies the state of the referent by defining a slot Most-Recently-Created for its referent. It also adds some slots to the meta-object of the new instance. Consequently, the object Dossier and its new instance My-Dossier will have the following structure, where the italics highlight the modifications made by reflective computation

```
<Dossier> =
   Type: <Object>
   Meta: <Annotating-Meta-#4323>
   Most-Recently-Created: <My-Dossier>

<My-Dossier> =
    Type: <Dossier>
    Meta: <Annotating-Meta-#3487> =
              Type: <Annotating-Meta>
              Referent: {(>>)}
              Author: <Symbol Pattie>
              When-Created: <List (8 4 86)>
```

Note that the slots Author and When-Created in the Meta of My-dossier are read-only reflective representations in the sense that the causal connection to the aspects they represent is not guaranteed by the KRS-system. This will be the case for all extra slots created for a meta-object.

The object My-Dossier has inherited the deviating meta-object. Instances of My-Dossier will also be created with this special method. If we want to prevent inheritance of a deviating meta-object, we have to override the meta-object of the instance with the default meta-object. For example

```
(defobject Your-Dossier
   (a Dossier
      (Meta (a Meta-Object
               (Referent (>>))))))
```

Similarly deviating meta-objects can be constructed for program-objects. Figure 14 illustrates a deviating message-meta-object. In particular, this deviating meta-object takes care that tracing information is printed during evaluation of the message.

```
<Tracing-Message-Meta> =
    Type: <Message-Meta>
    Eval-Method:
     <If-#6789> =
        Condition: <Message (>> Emptyp Slot-List Referent)>
        Then-Part: <Message (>> Target Referent)>
        Else-Part:
           <Action-Sequence-#2654> =
              Actions:
                <List
                  (<Message (>> Println of
                                 [String "Getting the filler of slot "])>
                   <Message (>> Print of (>> Last Slot-List Referent))>
                   <Message (>> Print of [String " for the object"])>
                   <Message (>> Eval Print-Method Meta Target Referent)>
                   <Message-#6704> =
                      Type: <Message>
                      Meta: <Tracing-Message-Meta>
                      Target: {<Form (krs:get-filler-of
                                         (>> Target Referent)
                                         (>> Last Slot-List Referent))>}
                      Slot-List:
                         {<Message (>> But-Last Slot-List Referent)>}>
```

Fig. 14. A deviating meta-object - 2.

When we now create a message that has this deviating meta-object, as in

```
(defobject My-Message
   (a Message
      (Meta (a Tracing-Message-Meta))
      (Target George)
      (Slot-List [List (Type Birth-Year)])))
```

and we ask this message for the filler of its Eval slot, the following information will be printed

```
Getting the filler of slot Birth-Year for the object <George>
Getting the filler of slot Type for the object <Number 1955>
```

finally the result

```
<Number>
```

is returned.

6.2.2. Default Implicit Reflection

The default reflective computation that is performed by an object is determined by the meta-objects of the primitive objects. 3-KRS not only incorporates a reflective architecture, but it is also a reflective language. The primitive objects represent the global interpreter of the language, and these objects are accessible and modifiable in a causally connected way. Consequently, the default reflective computation of 3-KRS can be modified dynamically.

If we redefine the meta-object of a primitive object, all objects in the system will be interpreted by this special interpreter. For example, if the object Object is redefined as

```
(defobject Object
   (Meta (an Annotating-Meta
            (Meta (a Meta-Object)))))
```

all new objects of the system will be created with two extra slots in their meta-object. However, in order to guarantee the causal connection and finiteness of the interpretation, the new meta-object of Object has to get the object Meta-Object as its meta-object (actually, there has to be an n such that the meta^n of Object is the Meta-Object object, i.e. the meta-object corresponding to the LISP implementation of the language). Which means that a default-meta-object will still be handled by the LISP implementation. Every object (except for default-meta-objects) will now be considered as having a special meta-object: the meta-object of an object is now (by default) a copy of the modified meta-object of Object.

This implies that every action in the system requires an extra level of interpretation. Objects with a default meta-object will require two interpreter steps: (i) one explicit step to interpret the object by means of the default meta-object and (ii) one implicit step to interpret the default meta-object of the object. Objects with a user-defined meta-object will now require three levels of interpretation: (i) one explicit step to interpret the object by means of its deviating meta-object, (ii) one explicit step to interpret the deviating meta-object with the default meta-object and (iii) one implicit step to interpret the default meta-object. Consequently, such global changes will affect the performance of the system substantially.

Similar global effects on the interpretation of programs can be obtained by modifying the meta-objects of the primitive program-objects. For example, if we redefine the Message object to be

```
<Message> =
   Type: <Program-Object>
   Slot-List: <List-#2987>
   Target: <Object-#7658>
   Meta: <Tracing-Message-Meta-Bis>
```

all message-objects will print tracing information when they are asked for the filler of the Eval slot. Again, simply giving the object Message the Tracing-Meta-Object defined above, would not work. In order to guarantee the causal connection and finiteness, the message-objects that are created in the Eval-Method of the object Tracing-Message-Meta have to be created with the Message-Meta object (the one corresponding to the LISP implementation) as their meta-object.

6.3. Explicit Reflection

An object may also explicitly prescribe the reflective computation that it wants to perform. The code in an object will at run-time cause reflective computation if it specifies computation about a meta-object which has as referent the object again. For example, the object-level code may include a message sent to a specific meta-object. The object Person in figure 15 shows an example. A Person object has an Annotating-Meta, which stores the time the Person object was created. In the Age slot, an explicit reference to the Meta of Person occurs: the When-Created slot of the Meta is used to compute the Age.

```
(defconcept Person
   (Meta (an Annotating-Meta))
   (Birth-Year (a Number))
   (Age (>> (Minus (>> Birth-Year))
            Year When-Created Meta)))
```

Fig 15. Explicit reflection.

The same granularity of reflective computation exists for explicit reflection. It is possible to specify explicit reflection for individual data-objects and program-objects as well as for the primitive objects of the language.

7. Related Work

7.1. Introduction

This section compares the 3-KRS experiment described above with related work. A first subsection stresses the contributions of 3-KRS to the state of the art in object-oriented languages. A second subsection situates the 3-KRS experiment in the context of the existing reflective architectures discussed earlier in this work.

7.2. Comparison with Existing Object-Oriented Languages

7.2.1. Introduction

When studying the existing literature on object-oriented languages, no object-oriented language incorporating a reflective architecture can be identified. However, this section shows that the introduction of reflective architectures is a logical step in the evolution of object-oriented languages.[2]

The notion of an object-oriented programming style evolved around 1970 with the development of the language SIMULA (Dahl & Nygaard,1966). SIMULA introduced two basic ideas of object-oriented programming, being:

(i) the notion of **object**, which is an entity combining data and program in that it has an internal local state and associated computation,

(ii) the notion of the **class** of an object, which determines the internal structure and the type of computation of the object. Classes are organised in a subclass hierarchy through which they may inherit information.

The important contribution of this new style was the new form of modularity it introduced in programming. A third basic idea of object-oriented programming which enforced this new form of modularity even more, was introduced by SMALLTALK-72 (Kay,1972):

(iii) computation in an object-oriented language comes from **message-passing**, i.e. instead of calling a procedure to perform an action on an object one sends the object a message. The selector of the message specifies the kind of action. Objects respond to messages using their internal data and programs.

Although the first object-oriented languages 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 languages. The important ideas behind object-oriented languages are **abstraction** and **encapsulation**. Many aspects of a computational system can be made local to objects. An object has a protocol for communicating with the world. It also has internal data, programs and computation which make that it can fulfill a certain computational role. However, these internal aspects are not accessible or visible to the outside, which means that an object can be free to realise its role in whatever way it wants to. Thus, an object could not only perform computation about its domain, but also about how it could realise this (object-) computation. The next section shows that designers of object-oriented languages have actually also felt the need to provide such facilities.

7.2.2. The Need for Reflective Facilities

Soon after object-oriented languages came into existence, the need for certain reflective facilities was felt. Two strong motivations existed. A first motivation was the need for specialised interpreters. It seemed 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 became clear that a specific design for an object-oriented language suited some applications, but was 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 actually also 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 was inspired by the development of **frame-based languages**, which introduces 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 seemed 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 activate specific procedures when specific events happen (e.g. the value becomes

instantiated or changed).

7.2.3. The Evolution Towards a Reflective Architecture

Object-oriented languages 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 16).

```
: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. 16. 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 with these languages 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. Consequently, meta-classes were introduced: a meta-class specifies the structure and computation of a class.

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 T. O'Shea (O'Shea,1986) 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 object-oriented languages 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 object-oriented languages 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 object-oriented languages tended 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 object-oriented languages 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 object-oriented languages were (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 object-oriented language with a reflective architecture differs from these languages.

7.2.4. Distinct Properties of 3-KRS

The important innovation of the 3-KRS language is that it fulfills the following crucial properties of an object-oriented reflective architecture[3]:

1. A first property is that it presents the first object-oriented language 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 reflective information that is available about the 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 as well as a meta-link to its meta-object.[4]

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, and to temporarily attach such a special behavior to an object (cf. next chapter).

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.

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 constructed on the basis of the real interpreter. 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) in the form of objects.

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 implementation action 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.

5. A last property is that the self-representation can also at run-time be modified, and these modification 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 because of the consistency of the self-representation result in actual modifications of the behavior of the system.

7.3. Comparison with Existing Reflective Architectures

Procedure-based, logic-based and rule-based languages incorporating a reflective architecture can be identified. Chapter III showed how procedural examples (variants of LISP) such as 3-LISP (Smith,1982) and BROWN (Friedman & Wand,1984) introduced the concept of a reflective function, which is just like any other function, except that it specifies computation about the current 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) were presented as 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.

We illustrated reflective architectures in rule-based languages by means of TEIRESIAS (Davis,1982) and SOAR (Laird, Rosenbloom & Newell,1986). These languages 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.

The 3-KRS experiment does for the object-oriented paradigm what the former languages did for the procedure, logic and rule-based paradigm respectively (cf. figure 17). Just like these languages, 3-KRS introduced a new concept (or programming-construct) being the notion of a meta-object.

Again 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.

SURVEY OF EXISTING REFLECTIVE ARCHITECTURES

	procedure-based	logic-based	rule-based	object-based
1.	code,env,cont circular int.	goal,clauses,etc. circular int.	rules,goal,state circular int.	state & behavior of objects & circular int.
2.	reflective functions	reflective theories	reflective rules	reflective objects
3.	meta-circular interpreter	sem. attach. refl. princ.	meta-circular interpreter	meta-circular interpreter

1. stands for the issue of self-representation
2. stands for the issue of how reflective computation is programmed
3. stands for the issue of causal connection

Fig. 17. A comparison of some existing reflective architectures.

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 (cf. chapter IV): 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 the language. For an object-oriented language these are: handling messages, creating instances, etc.

8. Conclusions

The purpose of this chapter was to illustrate the concepts introduced in chapter IV and V with a concrete example, being the implementation of the reflective architecture of the object-oriented language 3-KRS. The basic unit of information in 3-KRS is the "object". Data as well as programs are represented as objects in 3-KRS. An object groups information about the entity 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.

Objects by inheritance get a default meta-object. The default meta-object describes the standard implementation and interpretation of a 3-KRS object. It is a reification of the LISP implementation and interpretation of an object. An object can be given a special behavior by assigning to it a meta-object explicitly. The 3-KRS interpreter is meta-circular and object-oriented. The meta-object of an object is consulted whenever the interpreter has to perform some action on the object and the object has a deviating meta-object.

By means of this architecture every object has the possibility to show a reflective behavior. Reflective behavior can be obtained by giving an object a deviating meta-object or by accessing a meta-object from within the computation of an object. The slots of the meta-object of an object specify in an explicit way how the object behaves. At this level, the object itself and an explicit self-representation of its behavior are accessible to reason about or to make modifications.

This experiment represents the first reflective architecture in an object-oriented language. The existing object-oriented languages only support limited, ad-hoc reflective facilities, which leads to limitations and unclear designs, and consequently to problems in programming. However, over the years object-oriented languages have evolved towards designs providing more and more reflective facilities. So this experiment can be said to present the next logical step in this evolution.

The next chapter views the 3-KRS architecture from the programmers' point of view. It is shown that the architecture built, although far from trivial in implementation, actually provides a very elegant programming environment. Examples demonstrate how this architecture should be used and what tasks it facilitates.

NOTES

[1] Some object-oriented languages only provide a syntax for sending messages. An object is in these languages created by sending the message "New" to the type-object.

[2] It may be that the survey of object-oriented languages presented here is not complete. There are hundreds of existing object-oriented languages, mostly with very limited distribution and publication. We have selected the most well-known languages or those languages whose reflective facilities have been reported in the literature.

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

[4] 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.

CHAPTER VII

Programming in a Reflective Architecture

1. Introduction

The previous chapter illustrated the implementation of a reflective architecture for the language 3-KRS. This chapter uses the same language to illustrate what programming in a reflective architecture is like. Chapter I and II argued that reflective architectures are designed for a specific style of programming. The purpose of this chapter is to present a concrete demonstration of this style using 3-KRS.

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 a 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 object-oriented languages 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.

This makes it possible to temporarily associate a certain reflective computation with an object without having to change the object itself, which is for example very useful for debugging or tracing purposes. Another advantage is that libraries of reflective computation can be constructed. These libraries would consist of meta-objects representing a default-handling behavior, or a stepping and a tracing behavior, etc (cf. next section).

3-KRS has an architecture that supports the four uses of reflection discussed in chapter II. Since in 3-KRS data as well as programs are objects in the language, it supports not only reflection about data, but also reflection about programs. Further, 3-KRS is a reflective language, which makes it possible to obtain a general reflective behavior for all the data and programs of an application.

Note that 3-KRS represents the first language to unify reflection about data and about programs in one architecture. Most languages only support one of those. For example, 3-LISP only allows reflection about programs (reflective lambda expressions). In 3-LISP it is for example not possible to specify that a certain variable should have a reflective behavior.

Languages like SMALLTALK or COMMON-LOOPS on the other hand, only support reflection about data. Reflective behavior can only be obtained (and in limited ways) for the meta-classes of the system. Other languages do support both types of reflection but by means of different mechanisms. For example, TEIRESIAS supports several mechanisms for reflection: meta-rules (reflection about rules), schemata (reflection about data-objects) and templates (reflection about functions). However, these different mechanism are not integrated by a uniform design.

2. The 3-KRS Environment

Figure 36 shows the actual interface that is used when programming in KRS or in 3-KRS (Van Marcke,1987). 3-KRS incorporates a first version of a library of special-purpose meta-objects. The pane with label "tree 1" gives an overview of this library. It represents the inheritance hierarchy of all (named) meta-objects that are created. This hierarchy includes meta-objects for default-reasoning, sophisticated inheritance, frames, sophisticated printing, tracing, stepping, etc. Nevertheless, the development of a thought-out library of commonly needed meta-objects remains a topic for further research.

The meta-objects provided by the library are usable with very little effort or specific knowledge. It is possible to pick one "ready-made" meta-object and inspect its structure or definition. The bitmap shows how the object Default-Handler was "picked-up" and "put-down" in an editor pane. The result is that the definition of of the Default-Handler meta-object is shown. Default-Handler is an abstraction which can be instantiated every time an object with a default-reasoning behavior is needed.

The pane with label "lisp 1" shows how an instance of Default-Handler is created to give the object Bird a default-behavior. The only slot of the instance that remains to be filled is Defaults, which has to be a list of associations between slot-names and their default-fillers. When an instance of Bird is created, as in

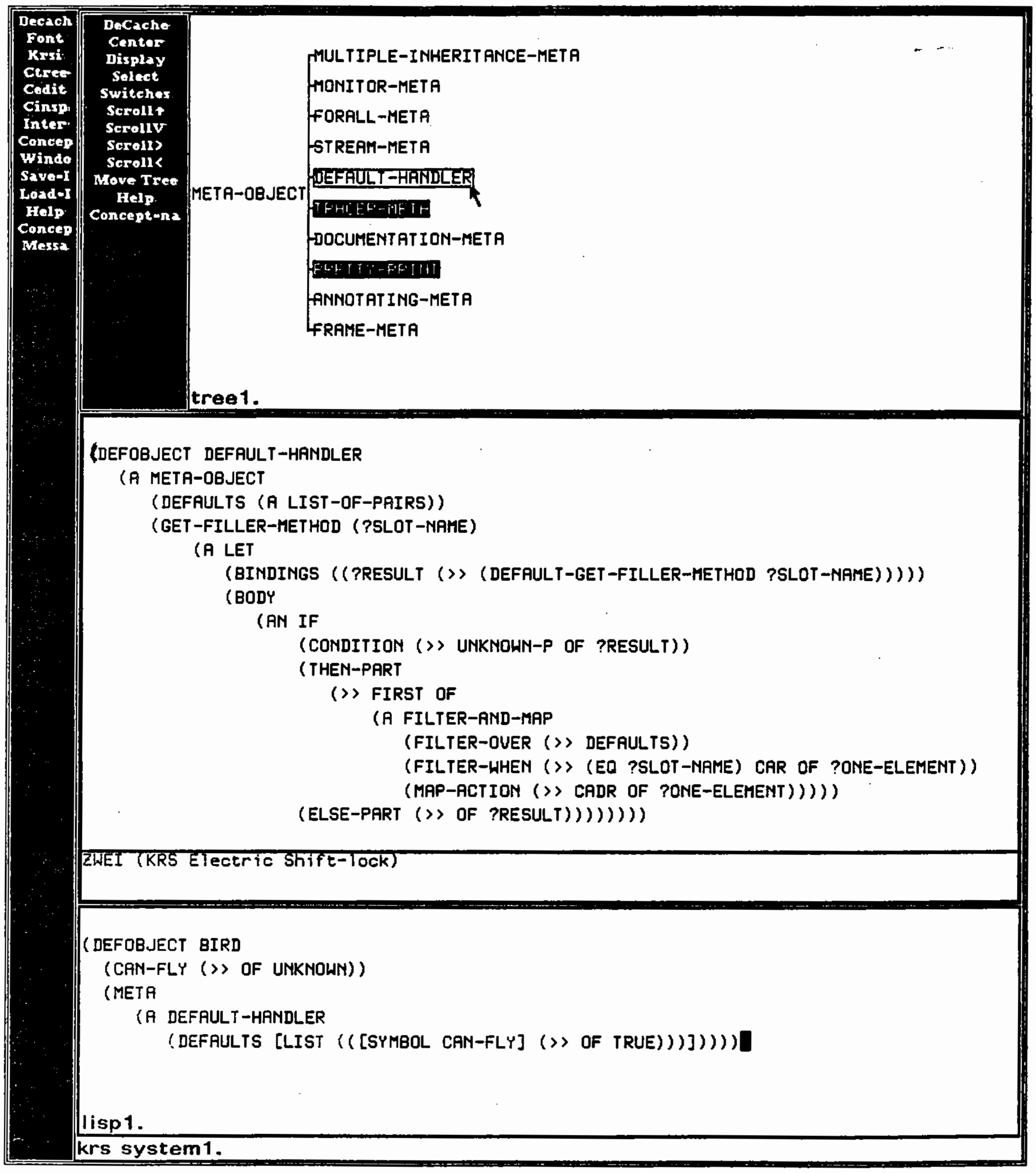
```
(defobject Tweety
   (a Bird))
```

and the message

```
(>> Can-Fly of Tweety)
```

is sent, the result will be the object

```
<True>
```



```
(DEFOBJECT DEFAULT-HANDLER
   (A META-OBJECT
      (DEFAULTS (A LIST-OF-PAIRS))
      (GET-FILLER-METHOD (?SLOT-NAME)
          (A LET
             (BINDINGS ((?RESULT (>> (DEFAULT-GET-FILLER-METHOD ?SLOT-NAME))))
             (BODY
                (AN IF
                    (CONDITION (>> UNKNOWN-P OF ?RESULT))
                    (THEN-PART
                       (>> FIRST OF
                           (A FILTER-AND-MAP
                              (FILTER-OVER (>> DEFAULTS))
                              (FILTER-WHEN (>> (EQ ?SLOT-NAME) CAR OF ?ONE-ELEMENT))
                              (MAP-ACTION (>> CADR OF ?ONE-ELEMENT)))))
                    (ELSE-PART (>> OF ?RESULT)))))))
```

```
(DEFOBJECT BIRD
  (CAN-FLY (>> OF UNKNOWN))
  (META
     (A DEFAULT-HANDLER
        (DEFAULTS [LIST (([SYMBOL CAN-FLY] (>> OF TRUE))]))))
```

Fig. 36. The actual 3-KRS environment.

3. Demonstration of the Use of Meta-Objects

3.1. An example Computational System

In order to demonstrate what programming in 3-KRS is like we have chosen an existing computational system. The example is a simplified version of the causal reasoning system described in (Van de Velde,1986). The purpose of the system is to make a diagnosis about a broken mechanical device. This section introduces the object-level code of the system. The next sections discuss how several interesting reflective computations can be attached to it.

The mechanical device is represented by means of a causal network. The nodes of this network can be mapped to properties of the device. They represent whether a certain property is Normal or Abnormal. For example, if the network represents the motor of a car, then the nodes in the network represent properties such as "Engine-Starts", "Cable-1=OK", etc. The arcs in the network represent causal links between nodes. If there exists an arc from node A to node B, this means that if node A is Abnormal, node B is also Abnormal. Figure 37 shows an example causal network (no values for the nodes are known yet).

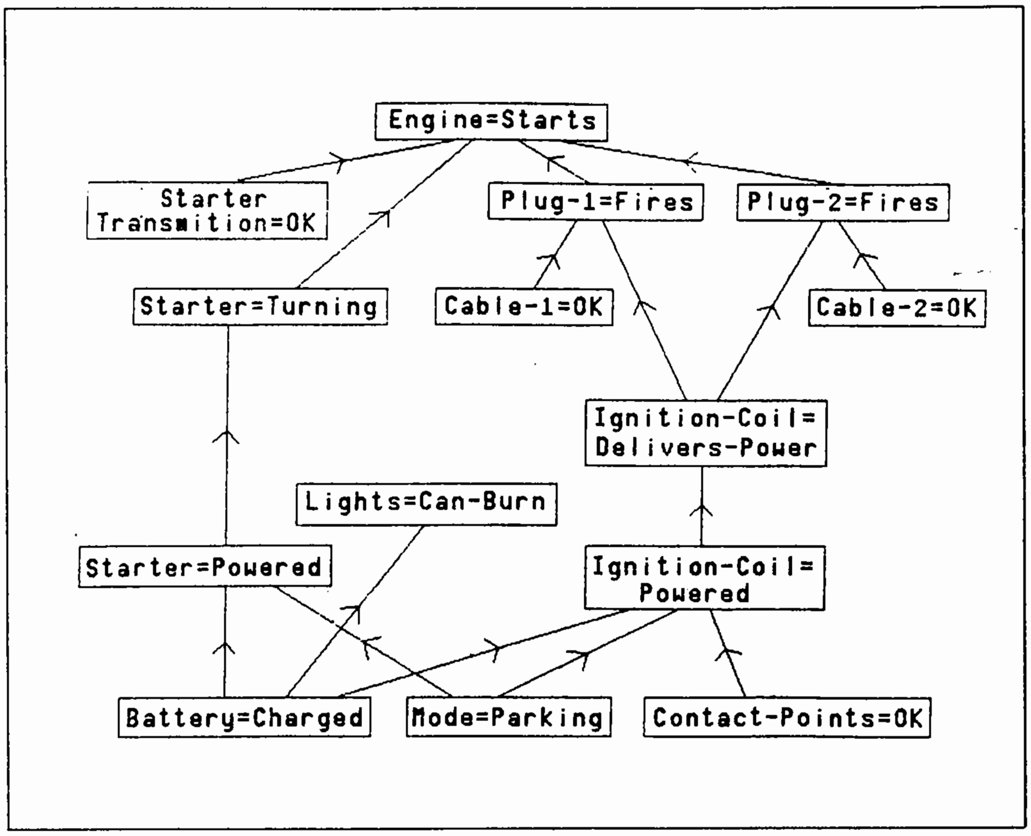


Fig. 37. A causal network.

This application introduces the objects of an Arc, a Node and a Diagnosis. An Arc has a slot From-Node and a slot To-Node, which are both filled by some node.

```
<Arc> =
  From-Node: <Node-#6478>
  To-Node: <Node-#8754>
```

An example of an arc from figure 37 is

```
<Arc-#9856> =
  From-Node: <Starter=Turning>
  To-Node: <Engine=Starts>
```

A node has a value. The value of a node can be Normal, Abnormal or Unknown. For some nodes the value is observable. If this is the case, we can ask the programmer for the value of the node by means of a pop-up menu. The slot Causes is filled by the set of nodes which are (direct) causes of the node. This set is computed by means of a Filter-and-Map object

(See (Jonckers,1987) for a more extensive description of the computational abstractions used in these examples).

```
<Node> =
   Value: {<If-#2121> =
             Condition: <Message (>> Observable-P)>
             Then-Part: <Message (>> Ask-Observed-Value)>
             Else-Part: <Message (>> of Unknown)>
   Observable-P: <False>
   Ask-Observed-Value:
      {<Query-Menu> =
          Query-String: <Fstring "~%Is ~S normal or abnormal ? (N//A) "
                                  (>> Contents Label)>
          Alternatives:
             <List (([String "Normal"] Normal [String "Normal")
                   ([String "Abnormal"] Abnormal [String "Abnormal"])))>}
  Causes:
     {<Filter-And-Map-#4321> =
          Filter-Over: {<Message (>> Forall of Arc)>}
          Filter-When: <Message (>> (Eq (>>)) To-Node of ?One-Element))>
          Map-Action: <Message (>> From-Node of ?One-Element)>}
```

When an Eval message is sent to a Filter-and-Map object, it selects those elements of the set in the slot Filter-Over that fulfill the predicate in the slot Filter-When. Afterwards, it maps the function in the slot Map-Action over the resulting set. The variable ?One-Element can be used to refer to one element of the set in Filter-Over. The predicate of this specific instantiation of Filter-And-Map states that the To-Node of an arc has to be equal to this node. The Map-Action function takes the From-Node of the arcs selected. For the above example, the message

```
(>> causes of Ignition-Coil=Powered)
```

will return

```
<Set (<Battery-Charged><Mode=Parking><Contact-Points=OK>))>
```

The object Diagnosis represents the diagnosis of a problem-node N. It has a slot Result which is filled by the defective nodes which cause the value of N. This set is computed by (i) making a diagnosis for every non-Normal cause of N, (ii) merging the resulting sets of defects in the slot Defects, (iii) adding the problem-node itself to the slot Defects in case it is abnormal and the set of Defects is still empty, and (iii) returning the set Defects.

```
<Diagnosis> =
   Problem-Node: <Node-#4343>
   Result:
     {<Action-Sequence-#2987> =
       Actions:
        <List
          (<Message (>> (Merge-with (>> Diagnosis-For-Causes)) Defects)>
           <If-#7980> =
               Condition:
                  <Message (>> (and (>> Empty Defects)
                                    (>> Abnormal-P Value Problem-Node)))>
               Then-Part: <Message (>> (Add (>> Problem-Node)) Defects)>
           <Message (>> Defects)>)>}
   Diagnosis-For-Causes:
      {<Filter-And-Map-Append-#3487> =
          Filter-Over: {<Message (>> Causes Problem-Node)>}
          Filter-When: <Message (>> Not Normal-P Value of ?One-Element)>
          Map-Action: <Message (>> Result of (a Diagnosis
                                               (Problem-Node ?One-Element)))>}
   Defects: <Set-#2121>
```

The slot Diagnosis-for-Causes is defined by means of a Filter-And-Map-Append object. This object first selects the causes of the Problem-Node whose value is non-Normal, and then applies the message

```
(>> Result of (a Diagnosis
                 (Problem-Node ?One-Element))
```

to each element of the resulting set. It appends the sets that result from this map-action. So finally the definition of Diagnosis-For-Causes returns the set of defects of all non-Normal causes of the problem-node.

If the causal-network looks as in figure 38, where A stands for Abnormal, U for Unknown and N for Normal, then

```
(>> Result of (a Diagnose
                 (Problem-Node (>> of Engine=Starts))))
```

returns

```
<Set (Mode=Parking Cable-2=OK)>
```

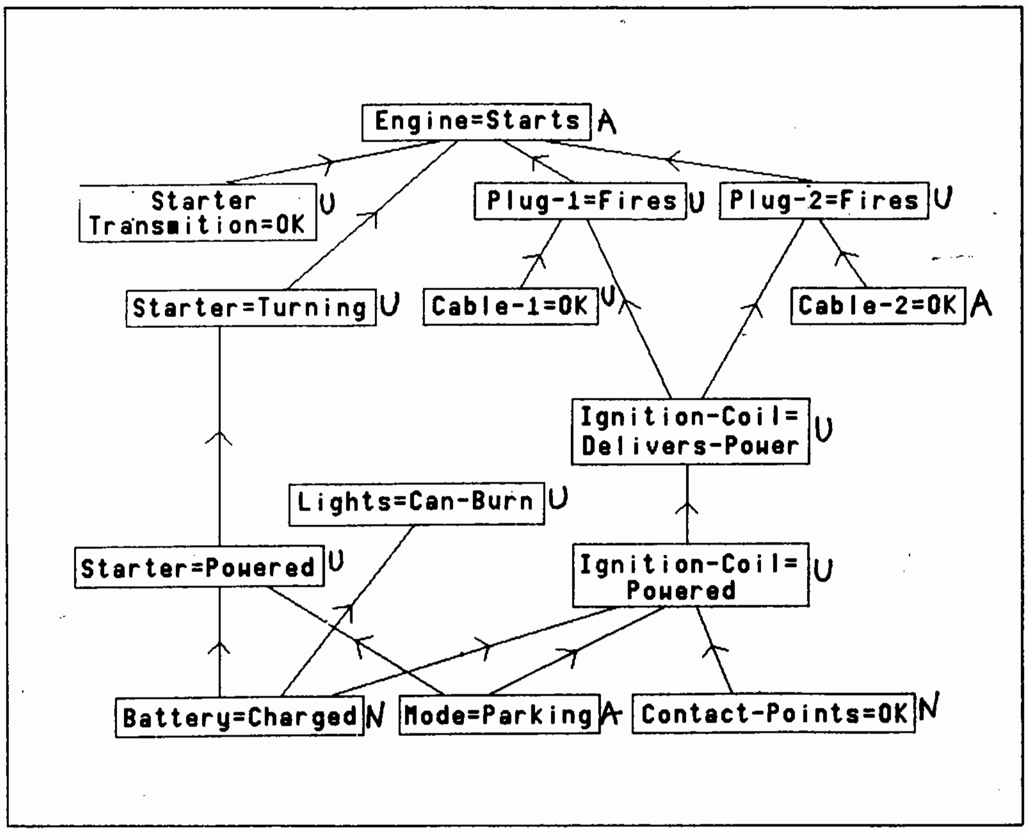


Fig. 38. The network to be diagnosed.

The following sections show how reflective computation can be introduced in this example computational system. The applications of meta-objects that are illustrated include:

(i) implementing local variations on the 3-KRS language,

(ii) implementing frames,

(iii) tracing,

(iv) improving the object-computation.

3.2. Variations on the Language

Although object-oriented languages have a long history, there is still no agreement on the fundamental principles of object-oriented programming. As it turns out the programming language community is still 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? how what messages look like? and what about message activation? etc.

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.

This section presents a simple example of a language variation for the causal reasoning application. The KRS language is designed such that the set of instances of a particular object is nowhere represented. It is not possible in the KRS language to ask for all objects of a specific type. Sometimes, it is of interset however to store this information. This is for example the case in the causal reasoning application. In the algorithm is is necessary to apply a Filter-And-Map over all instances of the Arc object (in italics)

```
<Node> =
  ...
  Causes:
     {<Filter-And-Map-#4321> =
          Filter-Over: {<Message (>> Forall of Arc)>}
          Filter-When: <Message (>> (Eq (>>))
                                     To-Node of ?One-Element))>
          Map-Action: <Message (>> From-Node of ?One-Element)>}
```

Rather than changing the basis instantiation mechanism of KRS, we can introduce a special-meta-object. The meta-object that was created for this purpose is presented in figure 39. Forall-Meta is a specialisation of Meta-Object that overrides the slot Make-Instance-Method, such that whenever an instance of the Referent has to be created, some special actions are taken.

The new make-instance-method first creates an instance using the default method. Then the slot First-Instance-P is asked for its filler. The definition of this slot specifies that a slot with name "Forall" has to be created for the Referent in case it does not exist already. The slot Forall is meant to be filled by the set of instances of the Referent. After that, the new instance is added to the Forall slot of the Referent, and finally the new instance is returned.

```
<Forall-Meta> =
  Type: <Meta-Object>
  Make-Instance-Method:
      <Let-#9876> =
         Bindings: <List ((?The-Instance
                             (>> Eval Default-Make-Instance-Method)))>
         Body:
           <Action-Sequence-#2789> =
               Actions:
                  <List (<Message (>> First-Instance-P)
                         <Message (>> (Add ?The-Instance)
                                      Forall Referent)>
                         <Message (>> of ?The-Instance)>)>
  First-Instance-P:
     {<If-#6578> =
        Condition: <Message (>> Not (Member [Symbol Forall]) Slots)>
        Then-Part: <Message
                       (>> Eval (Add-Slots [List ((Forall (a Set)))]))>}
```

Fig. 39. A variation on the default implementation of objects.

The meta of the Arc object inherits from this Forall-Meta object.

```
<Arc> =
  From-Node: <Node-#6478>
  To-Node: <Node-#8754>
  Meta: <Forall-Meta-#6543>
```

Consequently, whenever an instance of Arc has to be created, for example, when evaluating the sentence

```
(an Arc
    (From-Node (>> of Plug-1=Fires))
    (To-Node (>> of Engine=Starts)))
```

this will be handled by the meta-object of Arc, i.e. the slot Make-Instance-Method of Forall-Meta-#6543 will be evaluated. This method will take care that the slot Forall of the object Arc is updated (or possibly created, if this is the first instance). Finally the new instance is returned

```
<Arc-#2987>
```

So, when the message

```
(>> Forall of Arc)
```

is now sent, the set that is returned will contain the instance Arc-#2987. Note that the meta-object that we have created can immediately be used in other applications, as for example by

```
(defobject Person
   (Meta (a Forall-Meta
           (Referent (>>)))))
```

which creates a new object Person and gives it a Forall-Meta, or by

```
(>> (change-to (a Forall-Meta
                  (Referent (>>)))            *
    Meta of Person)
```

which alters the meta-object of the existing object Person. These make it possible to send messages such as

```
(>> Cardinality Forall of Person)
```

which returns the number of Person objects that have been created in the system (for * it only returns those instances that have been created after * was evaluated).

3.3. Frames

Chapter II discussed the notion of a "frame", which was introduced by the knowledge representation language tradition in order to facilitate the acquisition, maintenance and communication of knowledge structures. It showed that in most knowledge representation languages a data-item is a complex unit, called a frame. A frame not only contains a value, but also all sorts of reflective data and procedures about the value.

Until now the concept of a frame lacked a theoretical foundation. Most languages support a fixed set of facilities in a frame. Adding a new facility means changing the interpreter itself. For example, if we want to add a reflective facility to frames which makes it possible to specify how the frame should be printed, we have to modify the language-interpreter such that it actually uses this explicit print-method whenever the frame has to be printed.

In addition 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 a frame, it may no longer be clear wether the slot with name "Author" represents the author of the book (i.e. object data) or the author of the frame (i.e. reflective data).

In a reflective architecture, frames are viewed as an instance of reflection. Reflective knowledge and domain knowledge are in 3-KRS separated. The meta-object of an object contains all the refllective data and procedures for an object. A meta-object also "guards" and "steers" the object's computation.

Meta-objects can be used to implement frames in an explicit way. The library currently available in 3-KRS provides several frame-like meta-objects. We briefly discuss here the Monitor-Meta, the Stream-Meta, the Documentation-Meta and the Frame-Meta.

The Monitor-Meta can be used to trigger some computation when an object is created, touched, or changed. It also makes it possible to specify a constraint that will be checked everytime the object is modified. There is a slot If-Constraint-Fails which contains a program-object that is sent an Eval message when the evaluation of Constraint returned False, and a slot After-Constraint-Fails which tells whether the current computation should be proceeded or aborted after the constraint violation.

Figure 40 illustrates the Monitor-Meta. The fillers of slots represent the default-values that are used. Monitor-Meta overrides the Get-Filler-Method, the Inherit-Slot-Method and the Add-Slots-Method of the default meta-object. Basically, the new definitions add some actions before and after the default methods. These actions guarantee that the extra slots (e.g. Constraint, If-Created, etc.) are taken into account at the appropriate moments in the interpretation of an object.

```
<Monitor-Meta> =
   Type: <Meta-Object>
   Constraint: <Message (>> of True)>
   If-Constraint-Fails: <Form ()>
   After-Constraint-Fails: <Abort>
   If-Created: <Form ()>
   If-Changed: <Form ()>
   If-Touched: <Form ()>
   Get-Filler-Method(?Slot-Name): ...
   Inherit-Slot-Method(?Slot-Name): ...
   Add-Slots-Method(?Slot-Description-List): ...
```

Fig. 40. The Monitor-Meta.

The Stream-Meta makes it possible to specify what should happen when information is missing in an object. It provides facilities for defining the missing information, such as a default, a form to compute it and a rule-set to compute it. These different sources are explored in a given order when the information is absent. Figure 41 illustrates this meta-object. Stream-Meta overrides the Get-Filler-Method of the Meta-Object. It adds some actions which will compute the filler of a slot (using Default, Rules and To-Compute) in case the slot is empty.

```
<Stream-Meta> =
   Type: <Meta-Object>
   Default: <Unknown>
   Rules: <Rule-Set-#2376> =
            Rules: <List ()>
   To-Compute: <Form ()>
   Preference-List: <List (To-Compute Rules Default)>)
   What-With-Result: <No-Store>
   Get-Filler-Method(?Slot-Name): ...
```

Fig. 41. The Stream-Meta.

The Documentation-Meta records all sorts of reflective data about an object. It stores a documentation-string, when the object was created, by whom it was created, whether the object is modifiable or not, whether it is showable or not, etc. Figure 42 illustrates the Documentation-Meta. Documentation-

Meta overrides the slots Make-Instance-Method, Add-Slots-Method and Print-Method of the Meta-Object. The new definitions of these slots take care that slots such as When-Created and Author are filled, and that fillers of Showable and Modifiable are taken into account.

```
<Documentation-Meta> =
   Type: <Meta-Object>
   Documentation: <String "No documentation">
   When-Created: <List-#5498>
   Author: <Object-#9876>
   Showable: <Yes>
   Modifiable: <Yes>
   Make-Instance-Method: ...
   Add-Slots-Method(?Slot-Description-List): ...
   Print-Method: ...
```

Fig. 42. The Documentation-Meta.

The Frame-Meta is a combination of the Monitor-Meta, the Stream-Meta and the Documentation-Meta. The next example illustrates how the Frame-Meta can be used in the causal reasoning application. Suppose we want to record more information about the object Arc. This can be realised by giving the object Arc an instance of the Frame-Meta object. This specific instance overrides the slots Constraint, If-Constraint-Fails and Documentation. It also overrides the Make-Instance-Method such that instances are created the way a Forall-Meta creates them. Note that the special meta-object will be inherited by the instances of Arc (that do not override the Meta slot).

```
(defconcept Arc
   (Frome-Node (a Node))
   (To-Node (a Node))
   (Meta
      (a Frame-Meta
         (Referent (>>))))))
         (Constraint (>> Result of
                         (a Circularity-Check
                            (Start-Node (>> From-Node Referent)))
         (If-Constraint-Fails
             (>> Print of [FString "There is a circularity in the
                                   network for arc ~A"
                                  (>> Referent)]))
         (Documentation [String "An arc represents a causality
                                relation between two nodes of
                                a causal network"])
         (Make-Instance-Method (>> Make-Instance-Method
                                  (a Forall-Meta
                                     (Referent (>> Referent)))))))
```

The slot Constraint contains a message that will be sent everytime an Arc object is created or modified. This message checks whether the new Arc does not make the causal network circular. If it does, i.e. if there is a causality-path from the From-Node of the arc to the From-Node again, the filler of Constraint will be True. Consequently, the If-Constraint-Fails slot will be sent a message. This will cause the string

```
"There is a circularity in the network
for arc <Arc-#9870>"
```

to be printed. After that the computation will be aborted (because the inherited slot After-Constraint-Fails has filler Abort).

Note that because of the special meta-object, an Arc object will be much richer. A typical Arc object will look as follows

```
<Arc-#9856> =
   From-Node: <Cable-1=OK>
   To-Node: <Plug-1=Fires>
   Meta: <Frame-Meta-#9870> =
            Type: <Frame-Meta-#9807>
            Referent: <Arc-#9856>
            When-Created: <List (05 18 86)>
            Author: <Walter>
```

where

```
<Frame-Meta-#9807>
```

represents the meta-object of the object Arc.

Implementing frames by means of special meta-objects has many advantages over the classical frame-based languages:

- It provides a more modular solution, because the reflective information is separated from the domain-information of an object.
- It also provides a much more efficient solution. Objects that do not need a frame do not have this overhead in structure and computation. Classical frame-based languages store this extra information and perform this extra computation for every object of the language.
- The structure of a frame and frame-based reasoning is explicitly implemented, i.e. the semantics of a frame (i.e. an object which has as meta a Frame-Meta), become explicit and inspectable.
- Frames and frame-based reasoning are not only inspectable but also modifiable. It is possible to create new frame-style meta-objects from within the language itself. Other frame-based languages only provide a fixed design for frames.
- It is possible to specify procedural attachment for every operation that is performed on an object by the interpreter, printing, updating, retrieving, etc. Even more, it is not only possible to specify certain actions before and after an interpreter operation, but it is also possible to analyse and modify these operations.

3.4. Tracing

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 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 43 presents the Tracer-Meta object. This object is designed to be temporarily attached to a program-object. The Old-Meta slot has to be filled with the previous meta-object of the program-object. Tracer-Meta inherits from Program-Meta and overrides the Eval-Method. It adds some actions before and after the evaluation of the program-object, (the evaluationitself is still handled by the previous meta-object). These actions will take care that, when the program-object is sent an Eval message, some information is printed before and after the evaluation.

```
<Tracer-Meta> =
   Type: <Program-Meta>
   Old-Meta: <Program-Meta-#3761>
   Referent: <Program-Object-#8745>
   Eval-Method:
      <Action-Sequence-#7654> =
         Actions:
           <List
             (<Message (>> Print of
                                   [String "Starting the evaluation of: "])
              <Message (>> Snapshot Referent)>
              <Let-#8760> =
                 Bindings:
                    <List ((?result (>> Eval Eval-Method Old-Meta)))>
                 Body:
                   <Action-Sequence-#5454> =
                      Actions:
                         <List (<Message (>> Print of
                                                   [String "The result is: "])>
                                <Message (>> Snapshot of ?result)>
                                <Message (>> of ?result)>)>
```

Fig. 43. A meta-object for tracing programs.

A Tracer-Meta object can be temporarily attached to any program-object. Figure 44 presents two slots that were added to the Program-Object object in order to facilitates this. When a Trace message is sent to an object its meta-object is modified to a Tracer-Meta object. The slot Old-Meta of this Tracer-Meta object is filled with the previous meta-object. When an Untrace message is sent to a program-object, its Meta is set to its original meta-object again.

```
<Program-Object> =
   ...
   Trace: <Message (>> (Change-To (a Tracer-Meta
                                      (Old-Meta (>> Meta)))
                    Meta)>
   Untrace: <Message (>> (Change-To (>> Old-Meta Meta))
                      Meta)>
```

Fig. 44. The Trace and Untrace slot of a program-object.

These slots facilitate switching the tracing of a program on and off. For example, if we want to trace the definition of the slot Result of a Diagnosis object, this can be realised by sending the message

```
(>> Trace Definition Result of Diagnosis)
```

This message takes care that the definition of the slot Result gets a special meta-object (in italics)

```
<Diagnosis> =
   ...
   Result:
    {<Action-Sequence-#2987> =
      Meta: <Tracer-Meta-#8765> =
               Old-Meta: <Action-Sequence-Meta-#8333>
      Actions:
       <List
         (<Message (>> (Merge-with (>> Diagnosis-For-Causes))
                    Defects)>
          <If-#7980> =
              Condition:
                 <Message
                   (>> (and (>> Empty Defects)
                            (>> Abnormal-P Value Problem-Node)))>
              Then-Part: <Message (>> (Add (>> Problem-Node))
                                    Defects)>
          <Message (>> Defects)>)>}
```

Note that Action-Sequence-Meta-#8333 is the original meta-object of the Definition of Result. When the slot Result is asked for its filler, as in

```
(>> Result of (a Diagnosis
                 (Problem-Node (>> of Engine=Starts))))
```

the definition of Result is evaluated. Because this definition has a special meta-object, the slot Eval-Method of <Tracer-Meta-#8765> will handle the evaluation. Consequently information will be printed before the result is returned. If the tracing has to be switched off, this can be done by sending the message

```
        (>> Untrace Definition Result of Diagnosis)
```

which will reinstall the old meta-object.

3.5. Improving the Object-Computation

Learning behavior is by nature reflective. It requires a system to be able to reason and act upon its own object-computation. Learning can therefore most naturally be analysed and implemented by means of a reflective architecture. Figure 45 shows how simple learning capabilities are integrated in the causal reasoning application. The illustrated learning technique is a simpified version of the techniques adopted in second generation expert systems (Steels and Van de Velde,1986).

The object Diagnose modifies its computation by means of reflective computation. Actually it learns heuristics which alter its future search. Every time a Diagnosis object is asked for the filler of its Result slot, it will first try whether the heuristics that it already learned (represented by the set in Experienced-Defects) through its experience can explain the problem. If not, it will explore the complete search space (as it did before) and add the result of the diagnosis to the set of Experienced-Defects.

```
<Meta-For-Diagnosis> =
   Type: <Meta-Object>
   Referent: <Diagnosis>
   Get-Filler-Method(?Slot-Name):
      <If-#8760> =
         Condition: <Message (>> (Eq [Symbol Result]) of ?Slot-Name)>
         Then-Part:
            <Let-#5454> =
               Bindings:
                  <List ((?Successful-Heuristics (>> Shallow-Reasoning)))>
               Body:
                  <If-#9980> =
                     Condition: <Message (>> Not Empty
                                             of ?Successful-Heuristics)>
                     Then-Part: <Message (>> of ?Successful-Heuristics)>
                     Else-Part: <Message (>> Deep-Reasoning-And-Learning)>
         Else-Part: <Message (>> Eval Default-Get-Filler-Method)>
   Deep-Reasoning-and-Learning:
      {<Let-#7656> =
          Bindings:
             <List ((?New-Diagnose
                       (>> Eval (Default-Get-Filler-Method ?Slot-Name))))>
          Body: <Action-Sequence-#8787> =
                   Actions:
                     <List (<Message (>> (Add ?New-Diagnose)
                                         Experienced-Defects)>
                            <Message (>> of ?New-Diagnose)>)>
   Experienced-Defects: <Set-#9800>
   Shallow-Reasoning:
      {<Filter-#9998> =
         Filter-Over: {<Message (>> Experienced-Defects)>}
         Filter-With: <Variable ?One-Node-Set>
         Filter-When:
            <Message
               (>> Not (Member-P (>> of Normal)) of
                   <Mapping-#7776> =
                       Map-Over: {<Message (>> of ?One-Node-Set)>}
                       Map-With: <Variable ?One-Node>
                       Map-Action: <Message (>> Value of ?One-Node)>)>}
```

Fig. 45. Implementing learning in a reflective architecture.

The slot Experienced-Defects represents the set of sets of defective nodes which have already been identified as the cause of the problem in the past. The slot Shallow-Reasoning checks whether any of the Experienced-Defects already learned also applies to the current situation of the network. It returns those sets of nodes in Experienced-Defects for which all member-nodes are Abnormal. It computes this result by means of a Filter, which selects those sets in Experienced-Defects for which all their element-nodes are not Normal (which means that they are abnormal because they are observable).

The slot Deep-Reasoning-And-Learning searches the filler of the slot Result in the default way. This means that when Deep-Reasoning-And-Learning is asked for its filler, the search that is prescribed by the Result slot of Diagnosis will be performed. The Result slot returns the set of defects that causes the problem. This set is added to the slot Experienced-Defects afterwards.

Finally the slot Get-Filler-Method specifies that (i) first the slot Shallow-Reasoning has to be asked for its filler, (ii) is the resulting set (of sets) is not empty, this means that the defects that cause the problem were already encountered a previous time. Get-filler-method returns this set of defects without having used the definition of the slot Result. (iii) If the set of successful Experienced-Defects is empty, i.e. if none of the already learned Experienced-Defects applies to the current situation of the network, the slot Deep-Reasoning-And-Learning is asked for its filler. Finally, the other slots of the object Diagnosis will be handled in the default way.

When we attach this special meta-object to the object Diagnosis, as in

```
(defslot Meta of Diagnosis
         Meta-For-Diagnosis)
```

all Diagnosis objects will show a reflective behavior. Consider for example the object

```
<Diagnosis-For-Broken-Engine> =
    Type: <Diagnosis>
    Problem-Node: <Engine=Starts>
    Meta: <Meta-For-Diagnosis-#6545>
```

When this object is a first time asked for the filler of the Result slot, it will search the causal network for the defects that caused the node Engine=Starts to be Abnormal. Suppose it returns the set

```
<Set (<Cable-2=OK>)>
```

the next time Diagnosis-For-Broken-Engine is asked for its Result slot, it will first check whether it is not again the same Defect which caused the problem. If yes, the Result is immediately returned (without any causal reasoning happening). If not, the network is searched for the defects that this time caused the problem and these defects are added to the set of Experienced-Defects.

4. Conclusions

This chapter illustrated what programming in a reflective object-oriented language is like. It demonstrated the particular style of modular programming that is supported by reflective architectures. It argued that object-oriented reflection makes it possible to build abstractions of reflective behavior, which can be attached to an object whenever needed. The resulting programming environment is easy to use, flexible and efficient (as far as the reflective facilities are not exploited on a too broad scale). More examples of programming in 3-KRS can be found in (Maes,1986a), (Maes,1986b) and (Maes,1987).

Overall Conclusion

Computational reflection is a new approach for introducing modularity in programs. This work has made two original contributions towards a better understanding and utilisation of this approach. The more general contribution is that it has brought some perspective to various issues in computational reflection. A definition of computational reflection was presented, the importance of computational reflection was discussed and the design and construction of architectures that support reflection was studied. Illustrations were drawn from different sorts of languages including procedural, logic-based and rule-based languages.

The more specific contribution is that it has presented the first effort to introduce reflection in an object-oriented language. The implementation of a concrete reflective architecture was worked out and the programming style made possible by this architecture was extensively illustrated. The examples showed that a lot of programming problems that were previously handled on an ad hoc basis, can in an architecture for computational reflection be solved more elegantly.

The conclusions that can be made are that:

- Reflection can be defined in a clear and technical way.
- Reflection occurs frequently in real world programming, particularly in knowledge based systems.
- A reflective architecture provides better support for programming reflective computation.
- The following are the critical issues in constructing a reflective architecture: the self-representation, programming reflective computation and

the causal connection requirement.

- A reflective architecture is necessarily limited. The limitations, but also the capabilities, are determined by choices on these critical issues.
- A reflective object-oriented architecture can be realised and has specific advantages (e.g. grainsize, generality).
- The major advantage of a reflective architecture is that it allows the modular programming of reflective computation.

This work is only the beginning of a more systematic exploration of computational reflection. Some of the remaining open problems that can now be tackled are:

- Developing representations of computational systems which are interesting from a reflective point of view.
- How to avoid negative uses of reflection.
- Questions of efficiency.
- Building architectures which explore new areas in the design space of reflective architectures.
- The issue of reflective overlap and its relation to the self-referential paradoxes.
- Building libraries of reflective abstractions.
- Using reflection for building programming environments and program explanation facilities.

We hope this work will inspire the reader to reflect upon these issues.

Bibliography

Aiello L. and Levi G. (1984) "The uses of Metaknowledge in AI Systems". In: *Proceedings of The European Conference on Artificial Intelligence, ECAI 84*. Pisa, Italy.

Abelson H. and Sussman G. (1985) "Structure and Interpretation of Computer Programs". MIT-Press. Cambridge, Massachusetts.

Attardi G. and Simi M. (1984) "Meta-language and Reasoning across Viewpoints". In: *Proceedings of The European Conference on Artificial Intelligence, ECAI 84*. Pisa, Italy.

Barr A. (1977) "Meta-knowledge and Cognition". In: *Proceedings of The International Joint Conference on Artificial Intelligence, IJCAI 77*. Cambridge, Massachusetts.

Batali J. (1982) "Computational Introspection". Massachusetts Institute of Technology. Artificial Intelligence Laboratory. AI-MEMO 701. Cambridge, Massachusetts.

Bobrow D. and Stefik M. (1983) "The LOOPS manual". Technical Report KB-VLSI-81-13. Knowledge Systems Area, Xerox Parc, Palo Alto, California.

Bobrow D., Kahn K., Kiczales G., Masinter L., Stefik M. and Zdybel F. (1986) "COMMONLOOPS Merging COMMON LISP and Object-Oriented Programming". In: *Proceedings the Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 86*. Portland, Oregon.

Bobrow D. and Winograd T. (1977) "Experience with KRL-0, one Cycle of a Knowledge Representation Language". In: *Proceedings of The International Joint Conference on Artificial Intelligence, IJCAI 77*. Cambridge, Massachusetts.

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

Bowen K. and Kowalski R. (1982) "Amalgamating Language and Meta-language in Logic Programming". In: *Logic Programming.* Eds: Clark K. and Tarnlund S. Academic Press.

Breuker J. and Wielinga B. (1986) "Models of Expertise". In: *Proceedings of The European Conference on Artificial Intelligence, ECAI 86.* Brighton, Great-Britain.

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, ECAI 86.* Brighton, Great Britain.

Brownstow L., Farell R., Kant E. and Martin N. (1985) "Programming Expert System in OPS5". Addison Wesley, Reading, Massachusetts.

Bundy A., Byrd L., Luger G., Mellish C., Milne R., and Palmer M. (1979) "Solving Mechanics Problems Using Meta-Level Inference". In: *Proceedings of The International Joint Conference on Artificial Intelligence, IJCAI 79.* Tokyo, Japan.

Clancey W. (1983) "The Epistemology of a Rule-Based Expert System - A Framework for Explanation" In: *Artificial Intelligence Journal.* Volume 7, Number 3. North Holland. Amsterdam. The Netherlands.

Clancey W. (1986) "From Guidon to Neomycin and Heracles in Twenty Short Lessons". In: *AI magazine.* Volume VII, Number 3.

Clayton B. (1985) "ART: Programming Primer". Inference Corporation, Los Angeles.

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

Davis R. and Buchanan (1977) "Meta-level: Overview and Applications". In: *Proceedings of The International Joint Conference on Artificial Intelligence, IJCAI 77.* Cambridge, Massachusetts.

Davis R. and Buchanan (1984) "Meta-level Knowledge". In: *Rule-Based Expert Systems.* Eds: Buchanan B. G. and Shortliffle E. H. Addison Wesley.

Davis R. (1980) "Meta-rules: Reasoning about Control". In: *Artificial Intelligence Journal.* Volume 15, Number 3. North Holland. Amsterdam. The Netherlands.

Davis R. (1980) "Content Reference: Reasoning about Rules". In: *Artificial Intelligence Journal.* Volume 15, Number 3. North Holland. Amsterdam. The Netherlands.

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

Des Rivieres J. and Smith B. C. (1984) "The implementation of Procedurally Reflective Languages". XEROX Intelligent Systems Laboratory, ISL-4. Xerox Parc, Palo Alto, California.

Des Rivieres J. (1987) "Control-Related Facilities in LISP. In: *Meta-level Architectures and Reflection.* Eds: Pattie Maes and Daniele Nardi. North-Holland, Amsterdam, August 1987.

Doyle J. (1980) "A Model for Deliberation, Action and Introspection". Massachusetts Institute of Technology. Artificial Intelligence Laboratory. Technical Report 581. Cambridge, Massachusetts.

Ferber J. (1986) "Reflections in Object-Oriented Programming". In: *Proceedings of the International Workshop on Expert Systems and Their Applications, 1986.* Avignon, France.

Friedman D. and Wand M. (1984) "Reification: Reflection without metaphysics". In: *Communications of the ACM.* Volume 8.

Gallaire H. and Lasserre C. (1982) "A Control Meta-Language for Logic Programming". In: *Logic Programming.* Eds: Clark K. and Tarnlund S. Academic Press.

Genesereth M. (1983) "An Overview of Meta-Level Architecture". In: *Proceedings of the National Conference on Artificial Intelligence, AAAI-83.*

Genesereth M., Greiner R. and Smith D. (1980) "MRS Manual". Stanford Heuristic Programming Project. Memo HPP-80-24, Stanford University, Stanford, California.

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

Genesereth M. and Smith D. (1981) "Meta-level Architecture". Stanford Heuristic Programming Project, Memo HPP-81-6. Stanford University, Stanford, California.

Goldberg A. and Kay A. (1976) "SMALLTALK-72 Instruction Manual". Technical Report SSL-76-6. Xerox Parc, 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 University, Stanford, California.

Halpern J. (1986) Ed. "Theoretical Aspects of Reasoning about Knowledge. Proceedings of the 1986 Conference. Morgan Kaufmann.

Hayes P. (1974) "The Language GOLUX". University of Essex Report. Essex, Great-Britain.

Hayes-Roth et al. (1983) Eds. "Building Expert Systems". Addison Wesley.

Hofstadter D. (1981) "Meta-magical Themas". In: *Scientific American.* Number 244, 3.

Hunter G. (1973) "Metalogic: An introduction to the Metatheory of Standard First Order Logic". University of California Press, Berkeley.

IntelliCorp TM. (1985) "KEE TM. Software Development System. User's Manual". SEE Version 2.0. (Symbolics, LMI, Explorer), IntelliCorp.

Jansweijer W., Elshout J. and Wielinga B. (1986) "The Expertise of Novice Problem Solvers". In: *Proceedings of The European Conference on Artificial Intelligence, ECAI 1986.* Brighton. Great-Britain.

Jonckers V. (1987) "A Framework for Modeling Programming Knowledge". Ph.D. Thesis. VUB AI-LAB. Brussels, Belgium.

Konolige K. (1985) "A computational Theory of Belief Introspection". In: *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI 85.* Los Angeles, California.

Laird J., Rosenbloom P. and Newell A. (1984) "Towards Chunking as a General Learning Mechanism". In: *Proceedings of the National Conference on Artificial Intelligence, AAAI 84.* Austin, Texas.

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

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

Maes P. (1986a) "Introspection in Knowledge Representation". *Proceedings of the European Conference on Artificial Intelligence, ECAI 86*. Brighton, Great-Britain.

Maes P. (1986b) "Reflection in an Object-Oriented Language". *Proceedings of the SPL-Insight workshop*. Edinburgh, August 1986.

Maes P. (1987) "Object-Oriented Reflection" In: *Meta-Level Architectures and Reflection*. Eds: P. Maes and D. Nardi. North-Holland, Amsterdam, August 1987.

Maes P. and Nardi D. (1987) "Meta-Level Architectures and Reflection". North-Holland, Amsterdam, August 1987.

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

Mitchell T. (1983) "Learning and Problem Solving" In: *Proceedings of The International Joint Conference on Artificial Intelligence, IJCAI 83*. Karlsruhe. West Germany.

Moore R. (1977) "Reasoning about Knowledge and Action". *Proceedings of The International Joint Conference on Artificial Intelligence, IJCAI 77.*

Nardi D. (1986) "Evaluation and Reflection in F.O.L." In: *Meta-level Architectures and Reflection*. Eds: P. Maes and D. Nardi. North-Holland, Amsterdam, August 1987.

O'Shea T. (1986) "Why Object-Oriented programming Systems Are Hard to Learn". In: *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 86*. Portland, Oregon.

Perlis D. (1985) "Languages with Self-Reference 1: Foundations". In: *Artificial Intelligence Journal* Volume 25, Number 3. North Holland. Amsterdam. The Netherlands.

Perlis D. (1987) "Overview of Meta in Logic" In: *Meta-level Architectures and Reflection*. Eds: P. Maes and D. Nardi. North-Holland, Amsterdam, August 1987.

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

Shapiro E. (1983) "Algorithm Debugging". MIT Press. Cambridge, Massachusetts.

Silver B. (1986) "Meta-level inference." North Holland Publishing. Amsterdam, The Netherlands.

Smith B. (1986) "Varieties of Self-Reference" In: *Theoretical Aspects of Reasoning about Knowledge. Proceedings of the 1986 Conference.* Ed: Halpern J. Morgan Kaufmann.

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 Des Rivieres J. (1984) "Interim 3-LISP Reference Manual". XEROX Intelligent Systems Laboratory ISL-1. Xerox Parc, Palo Alto, California.

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

Steele G. and Sussman G. (1978) "The Art of the Interpreter". Massachusetts Institute of Technology, Artificial Intelligence Laboratory. AI-MEMO 453. Cambridge, Massachusetts.

Steele G. (1984) "Common-LISP: the Language". Digital Press.

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

Steels L. (1986b) "The Explicit Representation of Meaning". In: *Meta-level Architectures and Reflection*. Eds: P. Maes and D. Nardi. North-Holland, Amsterdam, August 1987.

Steels L. and Van de Velde W. (1985) "Learning in Second Generation Expert Systems". In: *Knowledge Based Problem Solving.* Ed: Kowalic. Prentice Hall, New Jersey.

Stefik M. and Bobrow D. (1986) "Object-Oriented Programming: Themes and Variations". In: *AI magazine.* Volume 6, Number 4.

Stefik M. "Planning and Meta-planning (MOLGEN: Part 2)". In: *Artificial Intelligence Journal.* Volume 16, Number 2. North Holland. Amsterdam. The Netherlands.

Sterling L. (1984) "Logical Levels of Problem Solving". In: *Proceedings of the Second International Logic Programming Conference.* Uppsala. Sweden.

Sussman G. (1982) "Implementing LISP". In: *Functional Programming and its Applications, an Advanced Course.* Eds: J. Darlington, P. Henderson and D.A. Turner. Cambridge University Press, London.

Van de Velde W. (1986) "Learning Heuristics in Second Generation Expert Systems". In: *Proceedings of the International Workshop on Expert Systems and Their Applications, 1986.* Avignon, France.

Van Harmelen F. (1986) "Improving the Efficiency of Meta-level Reasoning". Proposal for a Ph.D. thesis. July 1986. Edinburgh University.

Van Marcke K. (1986) "A Parallel Algorithm for Consistency Maintenance in Knowledge Representation". In: *Proceedings of the European Conference on Artificial Intelligence, ECAI 86.* Brighton, Great-Britain.

Van Marcke K. (1987) "Thesis Proposal". Vrije Universiteit Brussel. Artificial Intelligence Laboratory. Internal Report. Brussels, Belgium.

Van Melle W. (1980) "System Aids in Constructing Consultation Programs". UMI Research Press, Ann Harbor, Michigan.

Wand M. and Friedman D. (1986) "The Mistery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* Cambridge, Massachusetts.

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 Journal*. Volume 13, Number 1,2. North Holland. Amsterdam. The Netherlands.