西 南 交 通 大 学

本科毕业设计（翻译）

Java 反射机制研究与应用

年　　级：2011 级

学　　号：20112799

姓　　名：林从羽

专　　业：软件工程

指导教师：胡晓鹏

二零一五年六月

目　录

摘　　要 1
1. 简介 1
2. 什么是反射 2
3. 反射的作用 2
4. 什么是反射体系 3
5. 已有的反射体系 4
6. OOL 历史中的反射 6
ABSTRACT 8
1. Introduction 8
2. What is Reflection 9
3. The Use of Reflection 10
4. What is a Reflective Architecture 11
5. Existing Reflective Architectures 12
6. A History of OOL with Respect to Reflection 14
7. A Reflective Architecture in an OOL 18
8. A New Programming Style 20
9. Discussion and Conclusions 23
10. Acknowledgements 24
11. Bibliography 24

计算反射的概念与实践

Pattie Maes
A1-LAB
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
pattie@arti.vub.uucp

摘　要

本论文讨论了计算反射中各个方面的一些概念。论文给出了对计算反射的一个定义，讨论了计算反射的重要性，对一些支持反射的编程语言的体系进行了研究。然后，本论文给出了一个关于已有的反射实现尝试的概述，简要介绍了已有的具备反射体系的过程式、基于逻辑、基于规则的编程语言。本论文的主要部分描述了一个在面向对象的语言（Object-Oriented Language, 后文简称 OOL）中加入反射体系的首创尝试，强调了它对面向对象编程领域的贡献，阐述了一种新的编程风格。这个例子表明，很多以前通过临时的方式解决的编程问题，在一个反射系统中都可以更优雅地解决。

1. 简介

计算反射是这样的一种行为，它是由一个计算系统完成的关于自身计算的计算（并可能通过此行为影响原本计算过程）。虽然“计算反射”（下文简称“反射”）这个术语如今已经为人熟知，但与其相关的一些问题非常复杂，因此仍然有很多人对它的理解不够全面。论文的第一部分（第 2 到第 5 节）将阐明这一些问题，给出反射的一个定义并讨论它在编程中的应用。然后引入一个带有反射体系的编程语言的概念，这样的语言是为了支持反射而设计出来的。

反射体系在一些过程式[18]语言、基于逻辑的语言[24]、基于规则的语言[6]中已经有了实现。本论文的第二部分（第 6 到第 8 节）介绍了一个基于 OOL 的反射体系实现。现有的 OOL 语言仅支持有限的、临时支持的反射特性，这将导致一些设计受到限制且不清晰，进一步导致一些编程上的问题。不过，过去的几年中 OOL 的设计有了很多改进和发展，提供了越来越多的反射特性。论文讨论了这种发展趋势下可以预见的下一个阶段，讨论了一个在 OOL 中包含明确且统一的反射体系的原创尝试。这个实验展示了在一个 OOL 中实现一个反射系统是可能的，而且面向对象反射技术是有具体的优势的。

2. 什么是反射

本节将给出一个计算反射的定义，它对于任何反射模型来说都适用，无论是过程式的、演绎的、命令式的、消息传递式的或其他形式的实现。我们定义计算反射是一种由反射型系统——一个关于其自身并与其自身简单连系的计算系统——表现出来的行为。为使这个定义完整，接下来我们将对一些相关的概念进行讨论，比如计算系统、“关于”其自身的含义和简单连系。

计算型系统（下文简称“系统”）是一个基于计算机的系统，该系统作用在于回答其问题域（domain）的问题，并/或支持相关的行为。我们说系统是与其域相关的。它包含了用来表示该问题域自身的内部结构。这些结构包括用于表示域内实体及实体间关系的数据，以及一个规定这些数据应该如何被操作的程序。计算实际上是在处理器（解释器或者 CPU）执行这个程序（的全部或部分）的时候发生的[1]。所有的可执行程序都是计算型系统的一个例子。

如果一个系统与它的域是**简单连系**的，就意味着系统的内部结构和其描述的域是以这样的方式连接起来的：如果它们中的任何一者发生了改变，对另一者也会引起相应的影响。举个例子，一个控制着一根机械臂的系统，其内部存储了描述机械臂位置的数据结构。这些数据可能与机械臂的位置以这样的方式简单连系着：（i）如果机械臂在一些外力的作用下发生了移动，那么内部的数据结构也会发生相应的改变；（ii）如果内部数据结构（通过计算）发生了一些改变，那么机械臂也将移动到相应的位置。因此，一个简单连系着的系统永远都拥有关于其域的一个精确描述，并且系统可能通过它的计算引起该域的一些改变。

一个反射型系统是这样一个系统，它持有描述它自身（或自身的部分层面）的数据。我们称这些数据结构的总和为该系统的自我描述。这份自我描述使得系统能够回答关于其自身的问题并对其自身做出相应的行为。因为这份自描述的数据与其描述的系统层面是简单连系的，因此我们可以说：

（i）系统永远拥有一份其自身的精确描述；

（ii）系统的状态和计算与其描述总是保持对应。这意味着一个反射系统可以通过（对自身的）计算对自身做出修改。

3. 反射的作用

乍看之下，反射的概念似乎显得有点牵强。直到现在它依然被当成一个有趣且神

[1] 在一些语言中区分数据和程序的界线并不太清楚。即便如此，这不影响对我们在此提出的反射定义的理解。还有，对于一些语言来说，使用“推理”来替代“计算”可能会更加合适。

秘的特性，但却没人承认它的技术重要性。但是，我们认为，反射有很丰富的实践意义。计算领域的许多功能都需要反射的参与。大多数日常使用的系统展现出来的除了对象计算——即有关其外部问题域的计算——之外，也有反射计算，也即对其计算的计算。常见的反射计算有：保存性能统计数据、保存调试所需信息、调试和追踪特性、人机接口（比如，图形化输出、鼠标输入等）、关于下一步执行何种计算的计算（也称控制流分析）、自我优化、自我修改（比如，在机器学习系统中）以及自我激活（比如，通过监控器或后台驻留程序）等。

反射计算并不直接参与对系统外部域问题的解决。相反，它能帮助更好地进行系统内部组织或呈现系统内部与外部世界的接口。计算反射的目的在于保证对象计算过程中高效流畅的功能调用。

直到今天，编程语言并未充分认识到反射计算的重要性[2]。它们并未提供对其进行模块化实现的足够支持。举个例子，假设一个程序员想要跟踪一个计算，比如，在调试的时候，他一般必须在程序中添加额外的语句。调试完成以后，这些语句又必须从代码中移除，这通常又会引入新的错误。使用反射计算的场合在日常使用的计算系统中无处不在，编程语言应该把它当成一个基本的工具来支持。下一节我们将讨论编程语言如何来做到这一点。

4. 什么是反射体系

如果一个编程语言承认反射为一个基本的编程概念，并提供了处理反射计算的工具，我们就说这个编程语言支持反射体系。具体来说，以上论述将意味着：

（i）这种语言的解释器必须能为所有运行时系统提供描述了系统自身（或系统的部分层面）的数据。以这种语言来实现的系统就可以使用反射计算，只要包含解释这些数据应该如何使用的代码即可。

（ii）解释器同时也必须保证这些数据以及它们所描述的系统层面间的简单连系是完整的。如此，系统对其自描述做出的修改将反映到其自身的状态和计算上。

反射体系提供了思考计算系统的一个全新观念。在一个反射体系中，一个计算型系统被认为收纳了其对象部分和反射部分。对象计算的任务是去解决外部域的问题，提供其相应的信息，而反射层级的计算则着力于解决对象计算的问题并提供关于对象计算的相应信息。

在一个反射体系中，我们可以为程序临时地关联一个反射计算，比如，我们可以让这个计算在该程序解释执行的过程中同时对它进行追踪。假设我们要追踪一个规则

[2] 注意到一些更先进的编程环境可能已经提供了解决这里提到的一些问题的手段。但是，这些编程环境并不是以一个开放可拓展（open-ended）的方式来构建的，这意味着它们仅仅支持其中固定的几个反射特性。而且，它们通常仅仅支持静态的对计算的计算，也即，无法在运行时使用。

式系统的一个会话，并希望按照次序打印出应用在这个会话上的所有规则。在一个支持反射体系的编程语言中，这可以通过指定一个类似这样的反射规则来完成：

如果一个规则当前拥有最高的优先级
那么打印这条规则及与其相关的数据

而相同的情况如果是在一个不支持反射体系的规则式语言里，要达到相同的结果就只能通过修改解释器本身的代码（比如，让它打印出它所应用的所有规则的信息）或者修改所有的规则，让它们被应用的同时即打印出一些信息。

很明显，反射体系为实现反射计算提供了一个更加模块化的方法。众所周知，好的模块化可以使系统更容易管理、阅读、理解和修改。这还不是解耦合的唯一好处，更重要的是，它使引入实现反射型计算编程的抽象成为可能，就与诸如 DO 和 WHILE 这样的抽象控制结构对控制流编程的实现一样。

5. 已有的反射体系

这里将列举出一些支持反射体系的基于过程、基于逻辑以及基于规则的编程语言。3-LISP[18]和 BROWN[7]是过程式编程的两个例子（LISP 语言的变体）。它们引入了一个反射型函数的概念，这种函数与其他函数很像，除了一点，它描述的是对于当前正在进行的计算的计算。反射型函数是一种运行在解释器层级的局部函数：它对描述了代码、环境、以及当前正进行的对象层级计算的数据进行操作。

FOL[24]以及 META-PROLOG[3]是两个支持反射体系的逻辑式编程语言的例子。这些语言采用了元理论的概念。元理论和其他理论不同的地方在于，它进行推演的对象是另一个理论而非外部的问题域。一些谓词用于元理论中的例子有，“provable(Theory, Goal)”，“clause(Left-hand, Right-hand)”等。

TEIRESIAS[6]和 SOAR[13]是一些支持反射体系的规则式编程语言的例子。它们引入了元规则的观念。元规则与一般规则并无两样，除了它们指定的是关于正在进行计算的计算。这些规则操作的数据包含了诸如“推理过程出现僵局”“存在一个与当前目标相关的规则”“已撤销所有与当前目标有关的规则”等基本的规则元素。

如果我们仔细研究以上提到的反射体系，可以发现很多共性。其中一个即是，几乎所有这些语言都通过一个元循环解释器来实现其操作（FOL 中存在一个例外，我们稍后进行讨论）。元循环解释器是对语言的解释过程的一个描述，而这个解释过程本身就是用来运行这种语言的[3]。理论上，这种语言其解释过程需要无穷多个解释器，因为每个解释器都需要另一个解释器来解释它。技术上，这个无限循环可以通过引入一

[3] 这个描述最少应该包含这个解释器程序的名字（比如在 LISP 语言中就是“eval”）和其他的一些解释器数据，比如一个绑定列表和一个连续过程等。这些信息也可以更丰富，比如，可以提供更多关于解释器程序本身的信息。

个（以其他语言写成的）第二解释器来解决，这个解释器可以解释这整个解释器循环（当然，也必须保证产生与这个解释器循环相同的行为）。

所有这些反射体系都采用此种实现模型的原因是，元循环解释器展示了一种满足我们对简单连系的需求的方法。用来描述系统的自描述数据正是用来运行这个系统的元循环解释过程本身。因为这是系统的一个过程化描述，也即是说，以实现系统的程序自身来描述系统，我们称这些体系支持的是**过程化反射**。

在这种体系中，系统及其自描述之间的一致性是被自动满足的，因为这份自描述正是用来实现系统的数据。因此，简单连系的实现不是太大的问题。实际上仅存在着一份描述，它既用来实现整个系统，也用来推断和分析系统。注意，对于一个元循环解释器的实现来说，一个必要的条件是编程语言本身为语言和数据中的程序提供了一种通用的格式，更准确来说，可以认为这个程序是语言的数据结构和协议。

过程化反射存在的一个问题是，一份自描述被用来服务于两个目的。既然它作为服务于反射计算的数据，它就必须设计成可以被方便地用来分析系统的形式。但同时，这份描述它又被用来实现整个系统，这又意味着它必须足够高效。这通常是相互矛盾的设计需求。

因此，人们开始尝试开发一个不同的反射体系，使系统的自描述不再同时参与系统的实现。这种类型的体系被称为支持**声明式反射**的体系，因为它使得开发无需顾忌系统状态的自描述数据成为可能。这种状态，举个例子，可能是系统必须满足的时间或内存限制。这种自描述不必非得是系统的一种完整的过程化描述，它更像是一份约束的集合，指出系统必须满足的状态和行为。

在这种体系中，简单连接的需求要更难实现。它需要保证系统的外部表示和其内部获得的行为彼此之间保持一致。这意味着这种情况下，解释器自身需要决定系统如何与其自身的描述保持一致。所以某种程度上而言，解释器必须更加聪明。它必须能够把系统的声明式描述翻译成为实现系统的解释过程（也即过程化描述）。

可以认为这样的体系持有系统的两份不同形式的描述：系统的自描述及其所描述的系统本身。在计算过程中，更合适的描述会被选择使用。内部的（过程化）描述服务于系统的实现，而外部的（声明式）描述服务于对系统自身的计算。尽管在声明式的反射体系上可以开发出更多有趣的自描述实现，但这样的反射体系能够在多大程度上实现则仍然是一个未有结论的问题。GOLUX[12]和 Partial Programs[8]是两份值得注意的尝试。

实际上，声明式反射和过程化反射的区别应更多地看成是一个连续发展的过程。如 F.O.L.[24]这个语言则是处在这个过程的某个中间位置。F.O.L. 通过一种语意附着技术来保证自描述的准确性，而自描述的效用是通过反射原则来保证的。如果能证明这两种技术的结合可以成功地维护自描述和系统间的一致性，将有很大的实际意义。

6. OOL 历史中的反射

上一节讨论了现有反射体系的一些例子，分别是基于过程式、逻辑式和规则式的编程语言来实现的。接下来论文将把目光放在面向对象编程语言上。尽管最早的几个OOL，比如 SIMULA[5]或 SMALLTALK-72[9]，并不支持与反射计算相关的特性，但不得不说反射的概念天生与面向对象编程的精神一脉相承。OOL 一个重要的特性是抽象：系统中的对象可以以任意的方式完成它的任务。因此人们很自然地想到，一个对象不仅可以对其描述的问题域进行计算，同时它如何实现其（对象）计算也是一个问题。

实际上，OOL 的设计者们也感觉到提供这些特性的必要性，因有两个很大的需求。第一个需求来自专用解释器的设计。看起来在面向对象编程应该有什么基本原则上达成一致非常困难，编程语言社区对此也正在进行积极的讨论。他们尝试找出一些面向对象语言应该支持的“基本”特性[21]，如：区分类和实例对象是否必要？应该提供什么样的继承方式？消息看起来是什么样子的？等诸如此类的问题。

慢慢地一些事情变得清楚起来，特定设计的 OOL 可能适应于一些应用，而对于其他的应用则不适用。具备反射体系的语言则不会受到这些限制：反射使得为语言编写专用的解释器变得可能，并且可能仅使用这门语言本身做到这点。比如，对象可以被指定一个显式的、可修改的自描述，它可以描述对象信息的打印方式，或者实例对象的创建方式。如果这些显式的自描述与系统本身是简单连系着的（即对象的行为总是与自描述保持一致），那么对象就可能修改自身在这些方面上的行为。它们可以修改其自身信息被打印的方式，或者采取一种不同的过程来创建实例，等等。

第二个需求是随着基于构架的语言的发展而提出的。这种语言提出要把所有类型的反射数据和反射过程与问题域数据一起封装起来[16][17]。这样一来，一个对象就包含了不仅其所描述的域的信息，同时还有这个对象自身（实现和解释）的信息：对象何时被创建？由谁创建？它需要满足什么约束？等等。这些反射信息似乎在很多方面都有用武之地，比如：

(1) 它能为用户提供文档、历史记录和解释，以此来帮助他们应对一个巨大系统所带来的复杂度；

(2) 它能跟踪各描述之间的关系，比如一致性关系、依赖关系和约束关系等；

(3) 它能通过提供默认值或者值计算方法的方式来封装数据项的值；

(4) 它能保护数据项的状态和行为，并在特定的事件发生（比如值被实例化或者发生变化）时激活相应的过程；

一些 OOL 已经为这些需求提供了临时的方法进行反射，反射特性被混杂在对象层级的结构上。在如 SMALLTALK-72[9]和 FLAVORS[23]这样的语言中，对象不仅包含了它所描述之实体的信息，同时还包含了自描述，也即关于对象及其自身行为的信息。

举个例子，在 SMALLTALK 中，一个 Person 类可能包含一个用以计算该人的年龄的方法和一个控制 Person 类对象打印方式的方法。同样的，在 FLAVORS 语言中，每个 flavor 都包含了一组方法，这些方法记录了这个 flavor 可以使用的反射特性集合。

Concepts And Experiments In Computational Reflection

Pattie Maes

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

ABSTRACT

This paper brings some perspective to various concepts in computational reflection. A definition of computational reflection is presented, the importance of computational reflection is discussed and the architecture of languages that support reflection is studied. Further, this paper presents a survey of some experiments in reflection which have been performed. Examples of existing procedural, logic-based and rule-based languages with an architecture for reflection are briefly presented. The main part of the paper describes an original experiment to introduce a reflective architecture in an object-oriented language. It stresses the contributions of this language to the field of object-oriented programming and illustrates the new programming style made possible. The examples show that a lot of programming problems that were previously handled on an ad hoc basis, can in a reflective architecture be solved more elegantly[1].

1. Introduction

Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation. Although “computational reflection” (further on called reflection) is a popular term these days, the

issues related to it are very complex and at the moment still badly understood. The first part of the paper (sections 2 to 5) attempts to elucidate some of these issues. It presents a definition of reflection and discusses the use of reflection in programming. It further introduces the concept of a language with a reflective architecture, which is a language designed to support reflection.

Reflective architectures have already been realised for procedure-based[24], logic-based[24] and rule-based languages[6]. The second part of the paper (sections 6 to 8) discusses the realisation of a reflective architecture in an object-oriented language (further on called OOL). Existing OOLs only support limited, ad-hoc reflective facilities, which leads to limitations and unclear designs, and consequently to problems in programming. However, over the years OOLs have evolved towards designs providing more and more reflective facilities. This paper introduces the next logical step in this evolution. It discusses an original experiment to incorporate an explicit and uniform architecture for reflection in an OOL. This experiment shows that it is possible to realise a reflective architecture in an OOL and t hat there are specific advantages as well to object-oriented reflection.

2. What is Reflection

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

A **computational system** (further on called system) is a computer-based system whose purpose is to answer questions about and/or support actions in some domain. We say that the system is **about** its domain. It incorporates internal structures representing the domain. These structures include data representing entities and relations in the domain and a program prescribing how these data may be manipulated. Computation actually results when a processor (interpreter or CPU) is executing (part of) this program[2]. Any program that is running is an example of a computational system.

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

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

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

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

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

3. The Use of Reflection

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

Reflective computation does not directly contribute to solving problems in the external domain of the system. Instead, it contributes to the internal organisation of the system or to

its interface to the external world. Its purpose is to guarantee the effective and smooth functioning of the object-computation.

Programming languages today do not fully recognise the importance of reflective computation[3]. They do not provide adequate support for its modular implementation. For example, if the programmer wants to follow temporarily the computation, e.g. during debugging, he often changes his program by adding extra statements. When finished debugging, these statements have to be removed again from the source code, often resulting in new errors. Reflective computation is so inherent in every-day computation systems that it should be supported as a fundamental tool in programming languages. The next section discusses how languages might do so.

4. What is a Reflective Architecture

A programming language is said to have a **reflective architecture** if it recognises reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly. Concretely, this means that:

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

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

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

In a reflective architecture one can temporarily associate reflective computation with a program such that during the interpretation of this program some tracing is performed. Suppose that a session with a rule-based system has to be traced such that the sequence of

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

rules that is applied is printed. This can be achieved in a language with a reflective architecture by stating a reflective rule such as

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

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

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

5. Existing Reflective Architectures

Procedure-based, logic-based and rule-based languages incorporating a reflective architecture can be identified. 3-LISP[18] and BROWN[7] are two such procedural examples (variants of LISP). They introduce the concept of a reflective function, which is just like any other function, except that it specifies computation about the currently ongoing computation. Reflective functions should be viewed as local (temporary) functions running at the level of the interpreter: they manipulate data representing the code, the environment and the continuation of the current object-level computation.

FOL[24] and META.PROLOG[3] are two examples of logic-based languages with a reflective architecture. These languages adopt the concept of a meta-theory. A meta-theory again differs from other theories (or logic programs) in that it is about the deduction of another theory, instead of about the external problem domain. Examples of predicates Wed in a meta-theory are “provable(Theory,Goal)”, “clause(Left-hand,Right-hand)”, etc.

TEIRESIAS[6] and SOAR[13] are examples of rule-based languages with a reflective architecture. They incorporate the notion of meta-rules, which are just like normal rules except that they specify computation about the ongoing computation. The data-memory these rules operate upon contains elements such as

“there-is-an-impasse-in-the-inference-process”,

“there-exists-a-rule-about-the-current-goal”,

“all-rules-mentioning-the-current-goal-have-been-fired”, etc.

If we study the above mentioned reflective architectures, many common issues can he identified. One such issue is that almost all of these languages operate by means of a meta-circular interpreter (F.O.L. presents an exception which will be discussed later). A meta-circular interpreter is a representation of the interpretation in the language, which is also actually used to run the language[4]. Virtually, the interpretation of such a language consists of an infinite tower of circular interpreters interpreting the circular interpreter below. Technically, this infinity is realised by the presence of a second interpreter (written in another language), which is able to interpret the circular interpreter (and which should be guaranteed to generate the same behavior as the circular one).

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

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

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

Consequently, people have been trying to develop a different type of reflective architecture in which the self-representation of the system would not be the implementation

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

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

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

Such an architecture can be viewed as incorporating representations in two different formalisms of one and the same system. During computation the most appropriate representation is chosen. The implicit (procedural) representation serves the implementation of the system, while the explicit (declarative) representation serves the computation about the system. Although in architectures for declarative reflection more interesting self-representations can be developed, it is still is an open question in how far such architectures are actually technically realisable. GOLUX[12] and Partial Programs[8] are two attempts which are worth mentioning.

Actually the distinction between declarative reflection and procedural reflection should more be viewed as a continuum. A language like F.O.L.[24]is situated somewhere in the middle: F.O.L. guarantees the accuracy of the self-representation by a technique called **semantic attachment**. The force of the self-representation is guaranteed by **reflection principles**. It is far less trivial to prove that the combination of these two techniques actually also succeeds in maintaining the consistency between the self-representation and the system.

6. A History of OOL with Respect to Reflection

The previous section discussed examples of existing reflective architectures in procedure-based, logic-based and rule-based languages. We now turn to object-oriented languages. Although the first OOLs, such us SIMULA[5] or SMALLTALK-72[9], did not yet incorporate facilities for reflective computation, it must be said that the concept of

reflection tits most naturally in the spirit of object-oriented programming. An important issue in OOL is abstraction: an object is free to realise its role in the overall system in whatever way it wants to. Thus, it is natural to think that an object not only performs computation about its domain, but also about how it can realise this (object-) computation.

Designers of OOLs have actually felt the need to provide such facilities. Two strong motivations exist. A first motivation is the design of specialised interpreters. It seems to be very difficult to find an agreement on the fundamental principles of object-oriented programming. As it turns out the programming language community is still now actively experimenting in order to find the “basic” features an object-oriented language should support[21]: is a distinction between classes and instances necessary? what form of inheritance should be provided? what do messages look like? etc.

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

A second motivation is inspired by the development of frame-based languages, which introduced the idea to encapsulate domain-data with all sorts of reflective data and procedures[16][17]. An object would thus not only represent information about the thing in the domain it represents, but also about (the implementation and interpretation of) the object itself: when is it created? by whom is it created? what constraints does it have to fulfill? etc. This reflective information seems to be useful for a range of purposes:

- it helps the user cope with the complexity of a large system by providing documentation, history, and explanation facilities,

- it keeps track of relations among representations, such as consistencies, dependencies and constraints,

- it encapsulates the value of the data-item with a default-value, a form to compute it, etc.

- it guards the status and behavior of the data-item and activates procedures when specific events happen (e.g. the value becomes instantiated or changed).

OOLs have responded to this need by providing reflection in ad hoc ways. Reflective facilities were mixed in the object-level structures. In languages such as SMALLTALK-72[19] and FLAVORS[23], an object not only contains information about the entity that is represented by the object, but also about the representation itself, i.e. about the object and its behavior. For example, in SMALLTALK, the class Person may contain a method to compute the age of a person as well as a method telling how a Person object should be printed. Also in FLAVORS, every flavor is given a set of methods which represent the reflective facilities a flavor can make usage of (cfr. figure 1).

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

Fig. I. The structure cd the vanilla-flavor

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

A second problem is that they mix object-level and reflective level, which may possibly lead to obscurities. For example, if we represent the concept of a book by means of an object, it may no longer be clear whether 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[10]. 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[1]) is that it should also be possible to specify the internal structure and computation of a class. Meta-classes serve this purpose.

Meta-classes already made one improvement towards the distinction 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 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 Boming and O'Shea[2] reveals, users of SMALLTALK are often confused with meta-classes. We suggest that this confusion might well arise because of the undisciplined split between system information and domain information. A class in SMALLTALK is sometimes viewed as an object being an instance of a meta-class (i.e. as something containing reflective information), at other times it is viewed as a class containing information about the domain (i.e. representing an abstraction).

Another step towards the origin of reflective architectures was taken by the development of OOLs such as PLASMA[19]. ACTORS[14]. RLL[11] and OBJVLISP[4]. These languages try to bring more uniformity in object-oriented programming by representing everything in terms of objects. They all contribute to the uniformity of the different notions existing in OOLs by representing everything in terms of objects: class, instance, meta-class, instance-variable, method, message, environment and continuation of a message. This increased uniformity makes it possible to treat more aspects of object-oriented systems as data for reflective computation.

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

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

7. A Reflective Architecture in an OOL

This section discusses an OOL with an architecture for procedural reflection. The discussion is based on a concrete experiment that was performed to introduce a reflective architecture in the language KRS[20]. The resulting language is called 3-KRS[15]. The important innovation of 3-KRS is that it fulfills the following crucial properties of an object-oriented reflective architecture[5]:

1. A first property is that it presents the first OOL adopting a **disciplined split** between object-level and reflective level. Every object in the language is given a **meta-object**. A meta-object also has a pointer to its object. The structures contained in an object exclusively represent information about the domain entity that is represented by the object. The structures contained in the meta-object of the object hold all the reflective information that is available about the object. The meta-object holds information about the implementation and interpretation of the object (cfr. figure 2). It incorporates for example methods specifying how the object inherits information, how the object is printed, how a new instance of the object is made, etc.

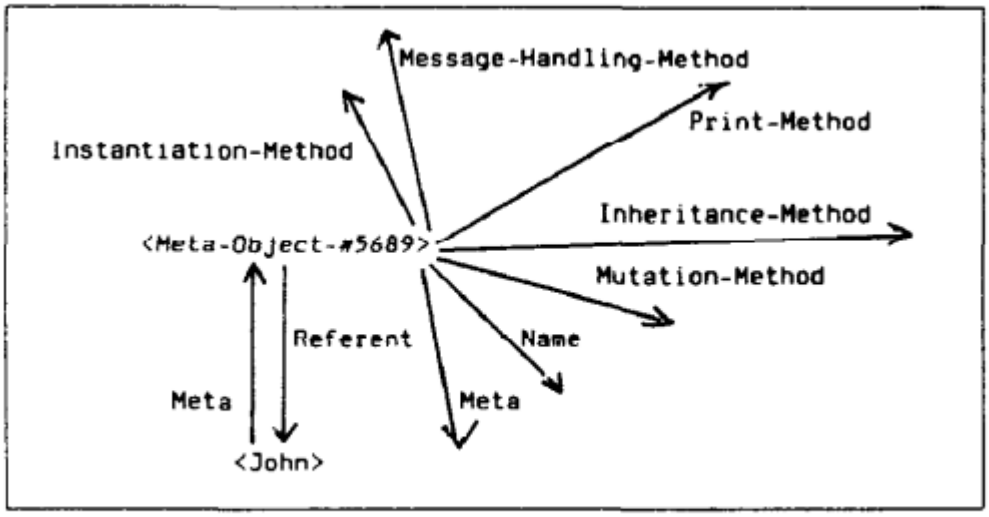

Fig. 2. An object and its meta-object.

Note that the meta-relation is not collapsed with the instance-relation (as it is in SMALLTALK-80 or LOOPS). The object John has a type-link to the Person object and a meta-link to its meta-object (named “Meta-Object-#5689”).[6]

[5] None of the languages discussed above fulfills the entire list, although they might fulfill one or more of the properties.
[6] 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.

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

2. A second property is that the self-representation of an object-oriented system is uniform. Every entity in a 3-KRS system is an object: instances, classes, slots, methods, meta-objects, messages, etc. Consequently every aspect of a 3-KRS system can be reflected upon. All these objects have meta-objects which represent the self-representation corresponding to that object. Note that since meta-objects are again objects, meta-objects have to be created in a lazy way. KRS incorporates a lazy-construction mechanism which takes care of this[22]: meta-objects are only constructed when they are actually needed.

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

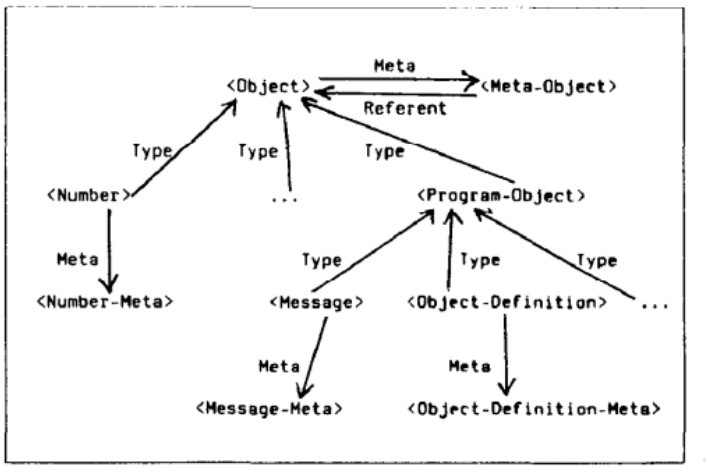

Fig. 3. The primitive meta-objects of 3-KRS, or the theory the language 3-KRS incorporates about the implementation of its objects and the interpretation of its programs.

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

4. A fourth property is that the self-representation of a 3- KRS system is **consistent**. The self-representation is actually used to implement the system. The explicit representation of the interpreter that is embedded in the meta-objects is used to implement the system. Whenever some operation has to be performed on an object, e.g. an instance of the object has to be created or the object has to answer a message, or a message-object has to be evaluated, the meta-object of the object is requested to perform the action. The technique that is used in order to avoid an infinite loop is that there is a second, implicit interpreter which is used to implement the default (or standard) behavior[7].

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

The 3.KRS experiment is extensively described in[15]. It shows that it is feasible to build a reflective architecture in an object-oriented language and that there are even specific advantages to object-oriented reflection. These advantages are a result of the encapsulation and abstraction facilities provided by object-oriented languages. The next section illustrates these advantages. It presents two examples of programming in an object-oriented reflective architecture.

8. A New Programming Style

Although the implementation of 3-KRS is far from trivial, from the programmer's point of view the language has a simple and elegant design. The basic unit of information in the system is the object. An object groups information about the entity in the domain it represents. Every object in 3-KRS has a meta-object. The meta-object of an object groups

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

information about the implementation and interpretation of the object. An object may at any point interrupt its object computation, reflect on itself (as represented in its meta-object) and modify its future behavior.

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

This reflective architecture supports the modular construction of reflective programs. The abstraction and encapsulation facilities inherent to OOLs make it possible to program object-computation (objects) and reflective computation (meta-objects) independently of each other. There is a standard message protocol between an object and its meta-object which guarantees that the two modules will also be able to work with each other[8]. This makes it possible to temporarily associate a certain reflective computation with an object without having to change the object itself. Another advantage is that libraries of reflective computation can be constructed. This section (schematically) illustrates what programming in a reflective OOL is like. It demonstrates the particular style of modular programming that is supported by reflective architectures. More (operational code) examples of programming in 3-KRS can be found in[15].

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

Figure 4 illustrates the idea. Message-#3456 is an object representing some message. It has a meta-object, called Message-Meta-#2342 which may be a copy of the default meta-object for a message or a user-defined specialisation of this. The Tracer-Meta object is designed to be temporarily attached to any program-object. The meta-link from the program-object to the old meta-object is temporarily replace by a meta-link to (a copy of)

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

the Tracer-Meta. Tracer-Meta-#8765 inherits from this old meta-object and overrides the Eval-Method: it adds some actions before and after the eval-method of the old meta-object (such that the evaluation itself is still handled by Message-Meta-#2342). These actions will take care that when Message-#3456 is evaluated, some information is printed before and after the evaluation.

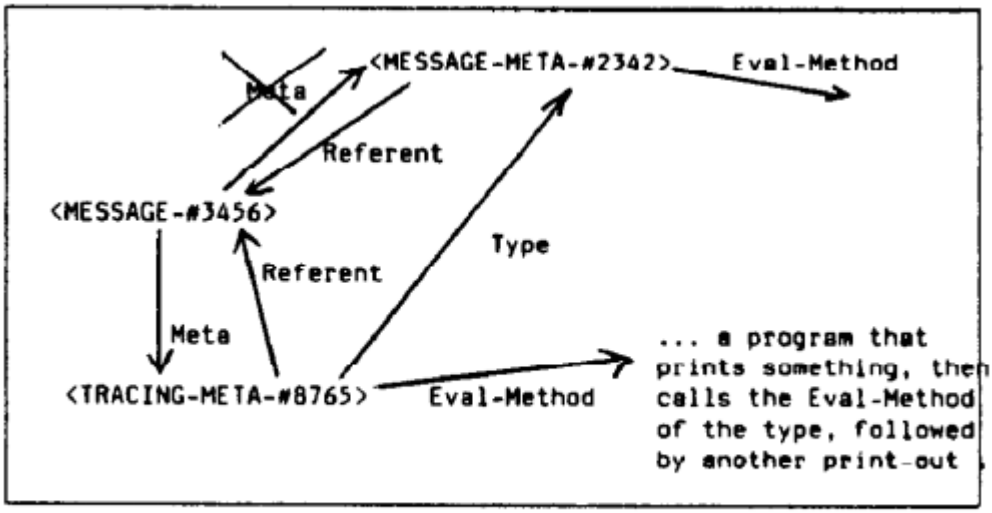

Fig. 4. Associating a tracing behavior temporarily.

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

A second example illustrates how a local deviating interpreter may be realised. A major advantage of a language with a reflective architecture is that it is open-ended, i.e. that it can be adapted to user-specific needs. But even more, a reflective architecture makes it possible to dynamically build and change interpreters from within the language itself. It allows for example to extend the language with meaningful constructs without stepping outside the interpreter. Note that this way the language itself can lx made more concise (and thus more efficient). The extra structure and computation necessary to provide objects with special features such as documentation, constraints or attachment do not have to be supported for all objects in the system but can be provided on a local basis.

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

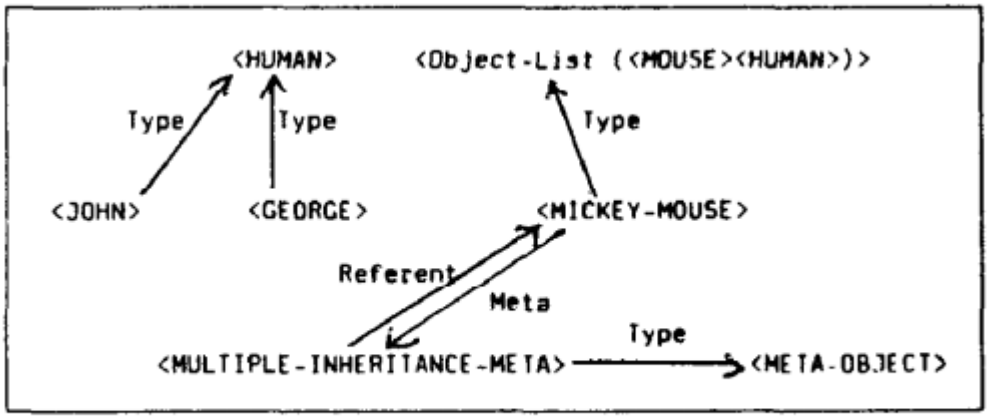

Fig. 5. Implementing a local variation on the language.

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

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

9. Discussion and Conclusions

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

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

This paper briefly introduced some of the concepts and experiments in computational reflection. However, many aspects of reflection, reflective architectures and particularly of object-oriented reflection (its implementation and use) have not been discussed in this paper. The interested reader may consult[15].

10. Acknowledgements

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

11. Bibliography

[1] Bobrow D., Stefik M. The LOOPS manual. Tech. Rep. KB-VLSI-III- 13. Knowledge Systems Area. Xerox Palo Alto Research Center. Palo Alto, California, 1981.

[2] Borning A., O'Shea T. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltak-80 Language. Proceedings of the ECOOP Conference. Paris, France, 1987.

[3] Bowen K. Meta-level Techniques in Logic Programming. Proceedings of the International Conference on Artificial Intelligence and its Applications. Singapore, 1986.

[4] Briot J.P., Cointe P. The OBJVLISP Model: Definition of a Uniform Reflexive and Extensible Object-Oriented Language. Proceedings of the European Conference on Artificial Inrellipcince. 1986.

[5] Dahl O., Nygaard K. SIMULA - An Algal-Based Simulation-Language. Communicadons of the ACM. 1966, 9: 671-678.

[6] Davis R. Knowledge-Based Systems in Artificial Intelligence. Davis R. and Lenat D. MC Graw-Hill, New York, 1982.

[7] Friedman D. and Wand M. Reification: Reflection without meta-physics. Communications of the ACM, Vol 8, 1984.

[8] Genesereth M. Prescriptive Introspection. Meta-Level Architectures and Reflection. P. Maes, D. Nardi. North.Holland. Amsterdam, June 1987.

[9] Goldberg A., Kay A. SMALLTALK-72 Instruction Manual. Technical Report SSL-76-6, Xerox Palo Alto Research Center. Palo Alto, California. 1976.

[10] Goldberg A. and Robson D. Smalltalk-80: The Language and its Implementation. Addison-Wesley. Reading, Massachusetts, 1983.

[11] Greiner R. RLL- I : A Representation Language Language . Stanford Heuristic Programming Project. HPP-80-9. Stanford, California, 1980.

[12] Hayes P. The Language GOLUX. University of Essex Report. Essex, United Kingdom, 1974.

[13] Laird J., Rosenbloom P., Newell A. Chunking in SOAR: The Anatomy of a General Learning Mechanism. Machine Intelligence. Vol 1. Nr 1. Kluwer Academic Publishers. 1986.

[14] Lieberman H. A Preview of ACT1. Massachusetts Institute of Technology, Artificial Intelligence Laboratory. MIT AI-MEMO 625. Cambridge, Massachusetts, 1981.

[15] Maes P. Computational Reflection. PhD. Thesis. Laboratory for Artificial Intelligence, Vrije Universiteit Brussel. Brussels, Belgium. 1987-1.

[16] Minsky M. A Framework for Representing Knowledge. Massachusetts Institute of Technology, Artificial Intelligence Laboratory. MIT AI-MEMO 306. Cambridge, Massachusetts, 1974.

[17] Roberts R., Goldstein I. The FRL Primer. Massachusetts Institute of Technology, Artificial Intelligence Laboratory. MIT AI-MEMO 408. Cambridge, Massachusetts, 1977.

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

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

[20] Steels L. The KRS Concept System". Vrije Universiteit Brussel. Artificial Intelligence Laboratory. Technical Report 86-l. Brussels, Belgium, 1986.

[21] Stefik M., Bobrow D. Object-Oriented Programming: Themes and Variations". Al magazine. Vol. 6. No. 4, 1986.

[22] Van Marcke K. A Parallel Algorithm for Consistency Maintenance in Knowledge Representation. Proceedings of the European Conference on Artificial Intelligence. Brighton, England, 1986.

[23] Weinreb D., Moon D. Lisp Machine Manual. Symbolics Inc. Cambridge, Massachusetts, 1981.

[24] Weyhrauch R. Prolegomena to a Theory of Mechanized Formal Reasoning". Artificial Intellipence Vol. 13 No. 1,2. North Holland. Amsterdam. The Netherlands, 1980.