# Implementing the Essence of Reflection: a Reflective Run-Time Environment

Massimo Ancona
DISI - Department of Informatics and Computer Science,
University of Genova, Genova, Italy
ancona@disi.unige.it

Walter Cazzola
DICO - Department of Informatics and Communication,
University of Milano, Milano, Italy
cazzola@dico.unimi.it

## ABSTRACT

Computational reflection provides the developers with a programming mechanism devoted to favorite code extensibility, reuse and maintenance. Notwithstanding that, it has not achieved developers' unanimous acceptance and its full potential yet. In our opinion, this depends on the intrinsic complexity of most of the reflective approaches that hinders their efficient implementation. The aim of this paper consists of defining the essence of reflection, that is, to identify the minimal set of characteristics that a software system must have to be considered reflective. The consequence is the realization of a run-time environment supporting the essence of reflection without affecting the programming language and with a minimal impact on the programming system design. This achievement will improve reflective system performances reducing the impact of one of the most diffuse criticism about reflection: low performance.

**Keywords:** Reflection, Run-Time Environment, Compiler Construction.

## 1. INTRODUCTION

Computational reflection [5] (or *reflection* for short) has been around for several years and many researchers have pointed out its potential and promises. However, it has never penetrated the programming realm at a level that we consider adequate for its capabilities. In our opinion, this is due to the absence of a simple characterization of the *real essence* of reflection that may lead to a *minimal* and orthogonal (that is, *independent of the programming language and paradigm*) implementation.

In this paper we present the minimal requirements to be satisfied by a programming framework in order to be reasonably considered reflective and to provide a very simple implementation mechanism. This set of requirements is the core of the reflective mechanism, what we call the *essence of reflection*. Our approach consists in embedding the essence of reflection into the run-time environment implementation, freeing the programming language from *reflective hooks*, i.e., clauses and linguistic constructs added to the language for supporting reflective programming. Moving reflective concepts from the programming language to its run-time environment has some surprising implications:

- Reflection has a minimal impact on the programming language clauses
- The implementation of compiler's front-end does not change from the original one. Changes needed to support reflection are moved in the compiler's back-end.

To validate our idea, we developed from scratch a simple programming system which embeds reflection from its early design stage. Such a language, inspired by Oberon [8], has been named lo[1] [1].


*SAC* '04, March 14-17, 2004, Nicosia, Cyprus


## 2. COMPUTATIONAL REFLECTION

### 2.1 Background on Reflection

Reflection is defined as the activity performed by an agent when doing computations about itself [5]. This activity involves two aspects: *introspection* (state and structure observation) and *intercession* (behavior and structure alteration). In a reflective system, entities (objects, processes, and so on) can be represented by other entities, usually referred to as *meta-entities*. Computation done by meta-entities (*meta-computation*) concerns the entities they represent, called *referents*. Meta-computations are often performed by *trapping* the normal computation of their referents. Of course, meta-entities themselves can be manipulated by meta-meta-entities, and so on. Thus, a reflective system can be structured in multiple levels, constituting a *reflective tower*. Base-level entities (termed *base-entities*) perform computations on the entities of the application domain. Objects in the other levels (termed *meta-levels*) perform computations on the entities residing in the lower levels.

*Reification* is an essential capability of all reflective systems. Each level of the reflective tower maintains a set of data structures representing (*reifying*) lower level aspects. Each reification must be *causally connected*[2] to the aspect(s) of the system being reified. All changes to the reification are reflected in the system, and vice versa.

*Transparency* is another key feature of all reflective systems. Transparency refers to the fact that the entities in each level are completely *unaware* of the presence and workings of entities in

[1]The name lo has two motivations: it comes from the Italian pronoun "io" ("the self" or "I myself") which is intrinsically "reflective" and from the name of a satellite of Jupiter which is a tribute to the programming language Oberon, that has strongly influenced lo architecture and which owes its name to a satellite of Neptune.

[2]A reflective system is causally connected when every change in the base-level system is reflected in the meta-level and vice versa.

higher levels. In other words, each meta-level is added to the base-level without modifying the referent level itself.

## 2.2 The Essence of Reflection

Meta-levels represent the autonomous components of a system, but their interface and relative interaction role are completely different and forms what we call the *essence of reflection*.

As stated in [2], the essence of every meta-computation can be summarized into two logical aspects: (i) the context switch between base- and meta-level, and (ii) the computation performed by the meta-entities.

The computational flow is in the base-level when a base-entity begins a computation. Then, such a computation is trapped by the corresponding meta-entity and the computational flow moves to the meta-level (*shift-up* action) where the meta-entity performs its computation. Finally, the computational flow goes back to the base-level (*shift-down* action) where the base-entity completes the interrupted computation. The computation performed by the meta-entity depends on the meta-program, and does not characterize the reflective mechanism. Instead, both shift-up and down actions are intrinsically tight to the concept of reflection itself and consent to introspect/intercede the behavior of the base-level system.

Shift-up and -down mechanism just allows behavioral introspection and intercession. Structural reflection is not captured by this mechanism. Structural reflection allows the manipulation of the system code and it is a feature characterizing a reflective system, and it is an integral part of our concept of essence of reflection. To carry out structural reflection, meta-entities need a reification of the base-level structure, that is, meta-entities must access directly to the code of the base-level or to a representative of such code.

From these considerations is fairly evident that the essence of reflection is more a system related feature than a programming language characteristic.

# 3. REFLECTION OUT OF THE LANGUAGE

Moving reflective mechanisms from the programming to the system level is a relatively recent trend [3, 7]. Most of the existing reflective languages support reflection through the introduction of *linguistic hooks*[3]. The main problems to be managed with this approach are:

- to minimize the execution overheads and performance penalties;
- to achieve a transparent use of reflection improving the reusability of the reflective features; and
- to avoid the use of ad hoc programming mechanisms and languages for the development of the base-level program.

Linguistic hooks are mechanisms to notify the system about when and where to apply a reflective manipulation. Moreover, it is necessary a MOP which provides a clear interface among levels allowing meta-entities to access and to alter structure and behavior of the base-entities. By using an implicit MOP, i.e., a MOP that doesn't need an explicit linguistic hook in the base-level code, the base-level's awareness about the existence of a meta-level is minimized.

[3]This means that we must introduce specific keywords in the base-level program to exploit reflection. By Example, in OpenJava [4] the programmer must specify which meta-entity manages each class through the `instantiates` keyword:

```
public class test instantiates TestMOP;
```

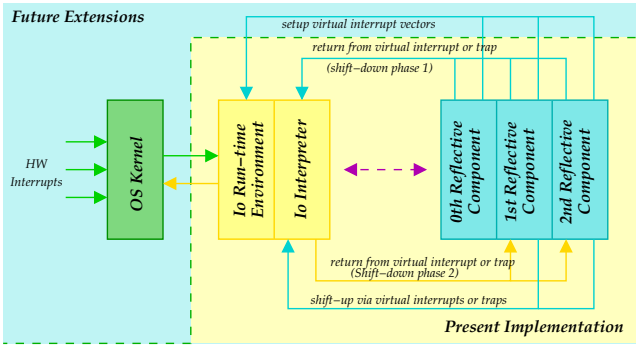


Figure 1: Reflective layout of the Io system

*Load-time reflection* permits of altering the program structure, during class loading whereas the other approaches apply their changes either at compile- or at execution-time. Javassist [3] is the most significant tool supporting load-time reflection in Java. It directly manipulates Java bytecode through a kind of reverse engineering which leads the substitution of specific bytecodes with bytecodes that are generated by the meta-program. Due to this approach context switching overheads are minimized, since there is not code to switch to and to be back from. As not expected, but welcome side effect there is that linguistic hooks are no more needed in the program code to support reflective actions.

As many other tools, Javassist is based on specific peculiarity of the language, i.e., Java's capability of dynamically changing the class loader, in order to achieve its goal. Therefore, load-time reflection is a good improvement for promoting reflection usability, but it is not a panacea because it is strongly tight to the language implementation.

The aim of this work consists of enhancing the load-time reflection approach by freeing it from implementation tricks and by merging the mechanism into the traditional structure of compilers in order to widen reflection applicability. Embedding the essence of reflection in the run-time environment seems to be the best approach for supporting both behavioral and structural reflection with small overheads, as we show in the rest of the paper.

# 4. EMBEDDING ESSENCE OF REFLECTION IN THE RUN-TIME ENVIRONMENT

Io [1] is an experimental, very simple and *apparently non reflective* programming language. Even if the Io programming language is free from reflective linguistic hooks, it permits reflective programming. This is, because we have embedded the essence of reflection in its run-time environment.

The ability of doing reflective programming with a nonreflective programming language proves that reflection is a system-wide feature and does not depend on the programming language [6].

The Io programming system is composed of a front-end which generates intermediate code, and two back-ends (a virtual machine and a x86 code generator) both working on the generated intermediate code.

Experimenting an interpreted language today has the strategic implication that the approach can be applied to important existing programming systems as Java and the .NET family. The use of a compiled language, on the other hand, by exploiting the dependencies of a real architecture makes the method efficient and usable in a general frameworks. In this paper, for sake of space, we just describe the interpreted approach.

The Io run-time environment provides the meta-level program with both a shift-up and -down mechanism (see section 4.2) and a complete access to the base-level structure (see section 4.1). For this reason, we state that Io implements the essence of reflection.

## 4.1 Structural Reflection

Structural reflection in Io is achieved by providing the meta-level program with a complete access to the memory layout of the programs in the underlying levels. This is done by exposing to the meta-level the data-structures of the Io interpreter containing the memory layout of the underlying levels. This data structure (a sort of array) reserves a slot to each level of the reflective tower.

The memory layout of each single level is composed of four segments: *segment table* (containing a sort of symbol table used to retrieving high-level information), *code segment* (containing, after the name, the code of the program), *state and register segment* (storing some information about the program execution, e.g., the program counter), and *working storage segment* (representing the memory used by the program).

The virtual storage layout is addressed via the display array. Level 0 of the display array is reserved to address the four segments of the memory layout of the controlled component(s). As the structural reflection is directly carried out on the memory layout of the controlled program the causal connection among base- and meta-level is granted.

The `IoENV` module — showed below — provides a high-level interface to the low-level details of the memory layout organization. `IoENV` must be imported by each component which wants to access to the structure of another component.

```
MODULE IoEnv(Io_Sys_);
(* This module opens all the low-level data structures of the Io in-
   terpreter implementing the controlled reflective component to the
   importing component. *)

OF Io_Sys USE Io_Word;

GLOBAL StrLng, SgtTop, CodeTop, WsBot, WsTop, pc_, ps_,
       dspy_, sgt_, code_, ir_, ...

CONST StrLng=...; SgtTop=...; CodeTop=...;
      WsBot=...; WsTop=...;
            ...
      TopNest=15; Layers=2; (* number of reflective layers - 1 *)

TYPE Order = RECORD op, a, b: INTEGER END;
     SgtTyp = ARRAY[0..SgtTop] OF SgtElTyp;
     CodTyp = ARRAY[0..CodeTop] OF Order;
     DspTyp = ARRAY[0..TopNest] OF INTEGER;
            ...

VAR sgt_: SgtTyp; (* the segment table *)
    code_: CodTyp; (* the code area *)
    ........
    t_, b_, hb_: INTEGER; (* tos, base and heapbase registers *)
    ........
    ir_: Order;     (* instruction buffer *)
    pc_, ps_: INTEGER;(* program counter and status registers *)
    .......
    dspy_: DspTyp; (* the display *)
    .......
    ws_: ARRAY[WsBot..WsTop] OF Io_Word;(* working storage *)
BEGIN
END.
```

Note that, the `Io_Sys` pragma addresses the reserved use of the zeroth level of the display.

## 4.2 Shifting-up and -down Mechanism

Reflection in Io is a mix of *component based* and *reactive* (event-driven) programming.

Component programming in Io is limited to the construction of the reflective program: each reflective layer is an independently executable program. Reflective components are independently compiled and implicitly interfaced via the shift-up and -down mechanism (see fig. 1).

The shift-up and -down mechanism is based on a *reactive model* supporting an event-driven programming mechanism in which the control flow is driven by *events*. Event-driven programming operates in response to events, and is based on a *dispatcher*, which activates the *event handlers*: small pieces of code encharged of performing the actions required by single events. Our events are: external events encoded into signals feed to the interrupt system hardware, program actions and computer actions.

The Io run-time environment (`IoEnv`) maps the reflective shift-up and -down mechanism over a *virtual interrupt management system* emulating a real processor architecture. Event-driven shifting is activated asynchronously by interrupt/trap signals sensed (or raised) by the virtual machine.

Essentially, the shift-up mechanism of Io promotes the computational flow at the meta-level on implicit execution patterns. These execution patterns are based on frequency (e.g., after each high level statement, at routine entry/exit or on external interrupts) and on class of constructs (e.g., at each loop statement, at each assignment and so on). How the shift-up takes place is specified via compilation switches.

The shift-down mechanism is completely under control of the meta-level program. A call to the predefined routine **`RETI(n)`** forces the control flow to shift down to level `n`. The control flow passes to the lower level component (which continues or starts its execution) and sets a special `intlev` variable of the run-time environment to `n`. The control flow will come back at the instruction after the **`RETI`** call, at the next shift-up operation having an interrupt `level` greater than or equal to `intlev`.

## 4.3 An Example: a Reflective Debugger

We conclude with a simple example showing how to program an event-driven reflective systems in Io. The example is relative to a simple line debugger called `DBX` able to debug whatever controlled component: it could be the starting point for implementing a general purpose reflective debugger or a sophisticated program tracer.

```
(* DBX is a simple line debugger. This module sets/resets breakpoints,
   changes and inspects global values and may execute the controlled
   program by single steps, routines entry/exit or breakpoints. *)

MODULE DBX;

OF DBXDefs USE StrngTyp,pc_,ps_,b_,t_,ShiftDown,PgmDmp,
   MiniDump,WriteTables,SetBreakPt,ResetBreakPt,ShowVar,
   ModVar,InitPgm;

VAR ch:CHAR; radr:INTEGER;id:StrngTyp;

(* dbx input main body *)

BEGIN
  WRITELN('dbx version 1.0');
  WRITELN('ps=',ps_:8,'pc=',pc_:8,'t=',t_:8,'b=',b_:8);
  DO
  WRITE('enter command [b,s,...]>');
  READLN(ch);
  CASE ch OF
```

```
    'b','B': (* break *)
       READLN(radr);
       IF radr>0 THEN SetBreakPt(radr)
       ELSE ResetBreakPt(-radr)
       FI\
    'x','X': WRITELN('exiting dbx'); HALT\    (* eXit *)
    'r','R': InitPgm\                        (* Restart *)
    's','S': WRITE('DBX: id>>>'); READLN(id);
             ShowVar(id)\                (* Show global value *)
    'm','M': WRITE('DBX: id>>>'); READLN(id);
             ModVar(id)\                 (* Modify global value *)
    'c','C','t','T','e','E','y','Y': ShiftDown(ch)
         (* reactivates pgm until next hook or breakpoint *)
    FO;
  OD
END.
```

DBX is activated first of the debugged program which starts its execution after the first ShiftDown procedure call. Then it takes the control after each active breakpoint, or upon routine entries/exits of after each statement execution, depending on the entered commands.

For illustrating the virtual interrupt system we disclose some details of module DBXDefs. DBXDefs implements high-level interface to the IoEnv module. DBXDefs imports from the system level module IoEnv all the data structure encoding a running component of the IO run-time environment. SetBreakPt and ResetBreakPt procedures access the code area for setting up interrupt priority of the chosen instructions (hooks implementation) for shifting-up activation/deactivation, while ShiftDown operates the effective shifting operation by calling the predefined routine **RETI** with the appropriate interrupt level masking. DBXDefs implements all low level interface used by DBX.

```
MODULE DBXDefs;
OF IoEnv USE StrngTyp,sgt_,code_,ir_,pc_,ps_,dspy_,...;

GLOBAL ShiftDown,WriteTables,SetBreakPt,ResetBreakPt,...;

CONST
     AllHooks   = 0; (* intlev for tracing *)
     EntRetHooks = 2; (* intlev for returns *)
     EntryHooks = 4; (* intlev for entries *)
     BreakHooks = 5; (* intlev for breakpoints *)

 PROCEDURE SetBreakPt(i: INTEGER);
 VAR j: INTEGER;
 BEGIN
   IF code_[i].op=hookop THEN
     WRITELN('pc:',i,code_[i].op,code_[i].a,code_[i].b);
     BreakNum:=BreakNum+1; (* incr. numb. of breaks *)
     code_[i].b:=BreakHooks; (* rise interrupt level *)
     WRITELN('pc:',i,code_[i].op,code_[i].a,code_[i].b);
   ELSE WRITELN('Not a legal point',i)
   FI
 END (* SetBreakPt *);


 PROCEDURE ResetBreakPt(i: INTEGER);
 VAR j: INTEGER;
 BEGIN
   IF code_[i].op=hookop THEN
     BreakNum:=BreakNum-1; (* decr. numb. of breaks *)
     code_[i].b:=AllHooks (* clear int level *)
   ELSE WRITELN('Not a legal point',i)
   FI
 END (* ResetBreakPt *);

 PROCEDURE ShiftDown(ch: CHAR);
```

```
BEGIN
  CASE ch OF
   't','T': RETI(AllHooks)\ (* traces each statements *)
   'e','E': RETI(EntryHooks)\ (* traces entry points *)
   'y','Y': RETI(EntRetHooks)\ (* traces entries and exits *)
   'c','C': RETI(BreakHooks) (* stops at ending or breakpoint *)
  FO; (* Here enters shift-Up!!! Examine program state *)
  IF ps_=running THEN
    WRITELN('DBX: program suspended at',pc_); MiniDump
  ELSIF ps_=ended THEN
    WRITELN('DBX: program terminated');HALT
  ELSE
    WRITELN('DBX: program aborted at',pc_);
    PgmDmp;HALT
  FI
END; (* ShiftDown *)
```

# 5. BENEFITS AND DRAWBACKS

This section is the complement of section 4, and should help the reader in dissolving his(her) doubts about the potentialities of our mechanism.

## 5.1 Is the Base-Level Free from Linguistic Hooks?

Usually, reflective programming languages adopt linguistic hooks in the base-level for informing the compiler/interpreter of:

- which associations bind each base-level entity to a specific entity in the meta-level;
- where the code of the corresponding meta-level entity can be retrieved from;
- when the execution flow passes from the base-level to the meta-level, i.e., the granularity of reflection [2].

There is no reason for hard-coding such information in the base-level program above all when the reflective mechanisms are provided by the run-time environment, and not by the programming language.

The IO framework provides a very fine and configurable granularity of reflection. On the basis of user choices, the execution flow may shift-up before executing each instruction of the controlled program or upon routine entry/exit.

The meta-level program is specified when the reflective system is launched and the associations, by default, are at program level. Such a loose coupling between base- and meta-level programs is the key idea that consents to free the base-level from linguistic hooks without reducing the generality of reflection.

Both the coupling and the scheduling of the shift-up and down actions can be regulated by an XML configuration file. This configuration file is read by the interpreter and used to configure the behavior of the reflective system on the fly. Moreover, given the particular architecture of the IO framework, the configuration steps can be executed by the meta-level program itself.

## 5.2 Is Our Approach Really Usable?

As shown in section 4.1, the IoENV module opens the structure of the controlled program up to the meta-level program. The access to each element (e.g., variables, procedures and so on) is achieved by using the low-level data structures of the interpreter, such as the symbol table and the execution stack and heap. A similar approach provides the meta-level programmer with a flexible and powerful mechanism which makes easier and faster to develop system-wide reflective applications such as self-debuggers and self-auditing tools. Unfortunately, the flexibility of the reflective mech-

anism and its implementation simplicity have the cost of reducing the readability of the meta-level program.

Programmers rarely need a so fine grain of structural reflection, and surely many of them are not at their ease in dealing with such a kind of details. To overcome this problem and to widen the framework usability we have written an external module (a library), named `IoReification`, which provides the meta-level programmer with a more abstract mechanism to access the structure of the base-level program than the `IoEnv` module. The `IoReification` module wraps the `IoENV` module (therefore it also wraps the inner representation of the base-level program) and provides a high-level API for manipulating it.

The `IoReification` module hides most of the low-level structural details providing the meta-level programmer with a course grain interface to structural reflection. A simple example of the approach used in `IoReification` is the procedure `ShiftDown` defined in the module `DBXDefs`, which encapsulates and hides the low-level data `pc_`, `ps_` and **`RETI`**. Our `IoReification` module is still simple but:

- it proves the flexibility of the approach which permits of building different abstraction levels starting from a low-level representation, and
- it renders the approach usable at different applications levels (e.g., system-wide, user own application and so on).

## 5.3 Efficiency Considerations

Our approach offers two advantages: simplicity of implementation and efficiency.

The simplicity of our method is measured by the augmented size of the reflective interpreter over the nonreflective one: it is only the 19% larger in number of source lines (8% for data definition and 22% for the instructions). The main reasons of this success are the *vectorization* of the data structures encoding the reflective tower into a simple array structure of the virtual machine and the extension of the display array (`dspy_[0]`) for having a fast direct access to the interpreter data structures from each reflective component.

About the execution time we have the following results:

- the overhead in execution time of nonreflective components with the reflective interpreter is only the 3% slower than that obtained with a nonreflective interpreter;
- The time for executing a monitored component with the maximum density of shift-up hooks (after each source level statement) is about the 11% of the time obtained without hooks. The controlling component is doing nothing (it returns the control by an immediate shift-down);
- The time for executing a monitored component with a medium density of shift-up hooks (after each routine/method entry) is about the 2% of the time obtained without hooks.

The above values are normalized with respect the execution of the same program and the same reflective interpreter with no hooks inserted.

# 6. CONCLUSIONS AND FUTURE WORKS

Embedding the essence of reflection in the run-time environment of a programming language has provided us with many advantages with respect to other reflective approaches. The most evident are the complete separation between base- and meta-level programs and a complete transparence of behavior. A less evident benefit is represented by the presence of a real reflective tower implemented inside the language back-end. This tower helps the complete separation of concerns and grants the access to the structural aspect of each program. Moreover, the mechanism allow a very flexible shift-up and -down mechanism with a finer and highly configurable granularity of reflection [2]. These characteristics allow building reflective components able to do computations about a general class of basic components e.g., reflective debuggers, with an extreme flexibility and efficiency.

Notwithstanding that the main concern about reflection still remains: how simple could be made the MOP in order to open up low level details (e.g., the virtual machine architecture and virtual code) without complicating their reflective manipulation?

In our opinion, this kind of gambling represents the most challenging difficulty on the way of a wide success and real usage of reflective mechanisms and systems.

Our method, however has the advantage to have clearly outlined the basic requirements and the core of every reflective system (the essence of reflection), thus simplifying the work that still has to be done. However this fact alone is not sufficient to grant success in diffusing the use of reflective paradigm.

At the end, we mention that the idea of using a prototypal framework (Io), which is not a toy, but it is still relative simple to support rapid prototyping and fast modifications, has been revealed essential in the process of understanding reflection and its basic mechanisms and relation with programming systems. Such a comprehension of reflection would be difficult to get looking at complex development frameworks, as Java and C++. The acquired know-how makes more simple to embed the essence of reflection also on these frameworks. Future works are just related to move the essence of reflection in the GNU back-end[4] and then rendering reflective the languages whose front-end supports the GNU back-end. .

# 7. REFERENCES


[1] M. Ancona and W. Cazzola. The Programming Language Io. TR DISI-TR-04-02, DISI, Università di Genova, May 2002.

[2] W. Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[3] S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *Proceedings of ECOOP'2000*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.

[4] S. Chiba, M. Tatsubori, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag, June 2000.

[5] P. Maes. Concepts and Experiments in Computational Reflection. In N. K. Meyrowitz, editor, *Proceedings of OOPSLA'87*, pages 147–156, Orlando, Florida, USA, Oct. 1987. ACM.

[6] J. M. Sobel and D. P. Friedman. An Introduction to Reflection-Oriented Programming. In *Proceedings of Reflection'96*, San Francisco, CA, USA, Apr. 1996.

[7] I. Welch and R. J. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, LNCS 1826, pages 157–169. Springer-Verlag, June 2000.

[8] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.


[4] GCC – GNU Compiler Collection project at `http://www.gnu.org/software/gcc/gcc.html`