The Oberon-2 Reflection Model and its Applications

Hanspeter Mössenböck, Christoph Steindl

Johannes Kepler University Linz
Institute for Practical Computer Science
Altenbergerstraße 69, A-4040 Linz
{moessenboeck,steindl}@ssw.uni-linz.ac.at

Abstract. We describe the reflection model of Oberon-2, a language in the tradition of Pascal and Modula-2. It provides run-time information about the structure of variables, types and procedures and allows the programmer to manipulate the values of variables. The special aspect of the Oberon-2 reflection model is that metainformation is not obtained via metaclasses. It is rather organized as structured sequences of elements stored on a disk, which can be enumerated by an iterator. This results in a simple and uniform access mechanism and keeps the memory overhead to a minimum. We also show a number of challenging applications that have been implemented with this reflection model.

1. Introduction

Metaprogramming, i.e. the observation and manipulation of running programs, has become an important instrument in the toolbox of today's software engineer. Pioneered by languages such as Lisp and Smalltalk, metaprogramming is now part of many modern programming languages such as Java [ArG96], CLOS [Att89] or Beta [LMN93]. If metaprogramming is not only applied to *other* programs, but also to the program that uses it, it is called *reflection*. A reflective program can obtain and manipulate information about itself.

In this paper we describe the reflection model of Oberon-2 [MöW91], a language in the tradition of Pascal and Modula-2. Oberon-2 is a hybrid object-oriented language. It provides classes with single inheritance that are declared within modules. Oberon [WiG89] is not only a programming language but also a run time environment, providing garbage collection, dynamic module loading, and so-called *commands*, which are procedures that can be invoked interactively from the user interface, thus providing multiple entry points into a system.

Commands and dynamic loading already constitute a kind of metaprogramming. True reflection, however, was added to Oberon-2 by the work of Josef Templ [Tem94]. We adapted and refined his ideas into a reflection model that allows us to answer questions like:

- What are the components of a record type *T* declared in a module *M*?
- What procedures are currently active? What are the names, types and values of their local variables?
- Does the caller of the currently executing procedure have a variable named x, and if so, what is its type and value?

Questions like these allow us to do a number of interesting things, which would not be possible with an ordinary programming language. We will show examples of such applications in Section 3 of this paper.

The rest of this paper is organized as follows. In Section 2 we describe the reflection model of Oberon-2 and its general usage. Section 3 shows a number of useful applications that we implemented on top of this reflection model, i.e. a generic output function, an object inspector, an embedded SQL facility, and an exception handling mechanism. In Section 4 we discuss security and performance issues. Section 5 summarizes the results.

2. The Oberon-2 Reflection Model

The reflection model of a programming language is characterized by two aspects:

- What metainformation is available about programs at run time?
- How can this information be accessed in order to observe and manipulate programs?

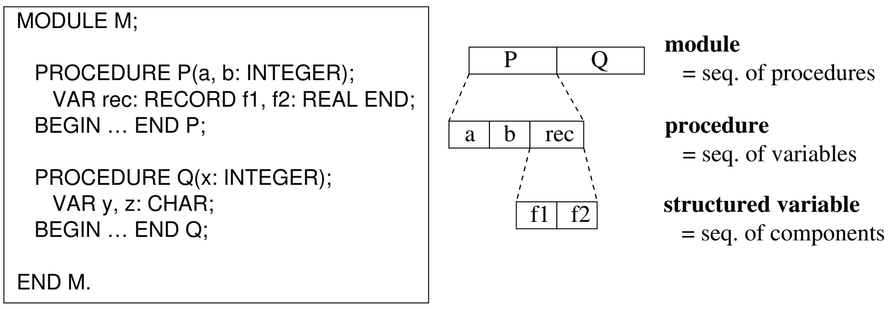
We will describe these two aspects for Oberon-2. The special thing about our approach is that all metainformation resides on disk instead rather than in main memory and that it is accessed by so-called *riders*, which iterate over the structure and parse it as required. This technique is space-efficient since no metaobjects have to be kept in memory. All access to the metainformation follows the *Iterator* pattern [GHJV95], which guarantees simple and uniform access.

Our reflection mechanism is encapsulated in a library module called *Ref* [StM96]. It defines a *Rider* type for iterating over the metainformation as well as procedures for placing riders on various kinds of metainformation sequences.

In the following sections we will first describe the structure of the metainformation and then explain how to navigate through it.

2.1 Metainformation

As a simple abstraction, a program can be organized as a set of hierarchical sequences. For example, a program consists of a sequence of modules. Every module is a sequence of variables, types and procedures. A procedure, in turn, is a sequence of variables, and so on. Fig. 1 shows an example of such a decomposition.



```
MODULE M;

   PROCEDURE P(a, b: INTEGER);
      VAR rec: RECORD f1, f2: REAL END;
   BEGIN ... END P;

   PROCEDURE Q(x: INTEGER);
      VAR y, z: CHAR;
   BEGIN ... END Q;

END M.
```

Fig. 1. Metainformation of module *M* in form of hierarchical sequences

Information about such program sequences is called *metainformation* and can be described by the following grammar in EBNF notation (curly brackets denote zero or more repetitions):

```
ProgramStruct = {Module}.
Module        = {Variable} {RecordType} {Procedure}.
RecordType    = {Variable} {Procedure}.
Procedure     = {Variable}.
Variable      = SimpleVar | RecordVar | ArrayVar.
RecordVar     = {Variable}.
ArrayVar      = {Variable}.

ProgramData   = GlobalVars | LocalVars | DynamicVars.
GlobalVars    = {Variable}.
LocalVars     = {Frame}.
Frame         = {Variable}.
DynamicVars   = {HeapObject}.
HeapObject    = {Variable}.
```

The information about the structure of a module is created by the compiler when the module is compiled. It is appended to the module's object file so that it is always copied and moved together with the object file. This avoids inconsistencies between a module and its metainformation.

The structural information is also used to interpret the program's data, which would otherwise be just a sequence of bytes without any interpretation.

2.2 Navigation

The metainformation is accessed by so-called *riders*, which are iterators that allow us to traverse a sequence of elements and to zoom into structured elements. The class *Rider* declared in module *Ref* looks as follows:

```
TYPE
  Rider = RECORD
    name: ARRAY 32 OF CHAR;
    mode: SHORTINT;    (*Var, Type, Proc, …, End*)
    form: SHORTINT;    (*Int, Char, Bool, Record, …*)
    ...
    PROCEDURE (VAR r: Rider) Next;
    PROCEDURE (VAR r: Rider) Zoom (VAR r1: Rider);
    ...
  END;
```

When a rider *r* is placed on an element of a sequence, *r.name* holds the name of this element; *r.mode* tells if the element is a variable, a type, a procedure, etc.; *r.form* encodes the type of the element (structured types are denoted by a special code; their components can be inspected by zooming into the element; see below).

Iterating. The following example shows how to traverse the global variables of a module *M* and print their names:

```
Ref.OpenVars("M", r);
WHILE r.mode # Ref.End DO
  Out.String(r.name);
  r.Next
END
```

A rider *r* is opened on the global variables of module *M*. While it is not moved beyond the last variable (*r.mode = Ref.End*) the name of the current variable is printed and the rider is advanced to the next variable by the operation *r.Next*.

Zooming. If a rider is placed on a structured element, it is possible to zoom into this element and to iterate over its components. For example, to access the local variables of the current procedure's caller we can zoom into the second frame on the activation stack, using the following statements:

```
Ref.OpenStack(r);
r.Next;
r.Zoom(r1)
```

Ref.OpenStack(r) opens a rider *r* on the frame of the currently active procedure. *r.Next* moves it to the caller's frame. *r.Zoom(r1)* zooms into that frame and sets a new rider *r1* to the first local variable in that frame. The variables of this frame can then be traversed as above using *r1*.

Placing a rider. Riders can be opened on various kinds of metainformation sequences as shown by the following table:

OpenVars(module, r)	sets *r* to the first global variable of the specified module
OpenStack(r)	sets *r* to the topmost stack frame
OpenPtr(p, r)	sets *r* to the first component of the object pointed to by *p*
OpenProcs(module, r)	sets *r* to the first procedure of the specified module
OpenTypes(module, r)	sets *r* to the first record type of the specified module

If a rider is opened on data (using *OpenVars*, *OpenStack* or *OpenPtr*), and if it is positioned on a non-structured variable, the value of this variable can be read or written using operations such as *r.ReadInt(n)* or *r.WriteInt(n).* In this case the rider serves as a link between the data (e.g. a stack frame) and the metainformation that is used to interpret that data (Fig.2).

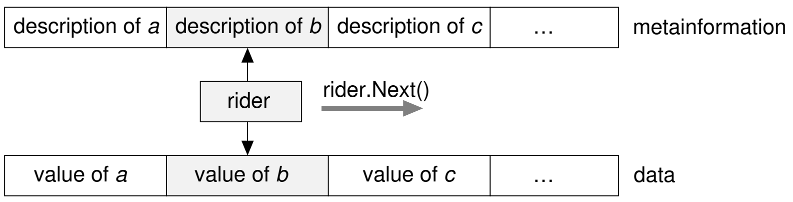


Fig. 2. A rider as a link between data and its metainformation

If a rider is opened on structural information (using *OpenProcs* or *OpenTypes*), there is no data to be read or written. Such riders can only be used to explore the structure of procedures and types.

Details about the *Rider* class, its fields and its operations are described in [StM96]. The difference between our implementation and the one in [Tem94] is mainly that we use a single rider type to iterate over all kinds of metainformation while [Tem94] uses special rider types for variables, procedures, types, etc.

2.3 Examples

The following examples should give you a rough impression of what you can do with the module *Ref* and its riders.

Assume that we want to print the names of all currently active procedures together with the names of their local variables. The following code fragment does the job:

```
VAR r, r1: Ref.Rider;
...
Ref.OpenStack(r);      (*r is on the most recent frame*)
WHILE r.mode # Ref.End DO
  Out.String(r.mod);   (*name of this frame’s module*)
  Out.String(".");
  Out.String(r.name);  (*name of this frame’s proc.*)
  Out.Ln;
  r.Zoom(r1);          (*r1 is on first var. of frame*)
  WHILE r1.mode # Ref.End DO
    Out.String(r1.name); Out.Ln;
    r1.Next
  END;
  r.Next               (*move to the caller's frame*)
END
```

Of course we could do any processing with the traversed variables or procedures. For example, we could print their values and types (this was actually used for the implementation of the Oberon debugger). We could also look for all occurrences of a certain value within the variable sequence and report them to a client.

The next example looks for a global record variable named *varName* declared in a module named *modName*. Note that the names of the variable and the module need not be statically known. They could have been obtained at run time.

```
VAR
  r: Ref.Rider;
  varName, modName: ARRAY 32 OF CHAR;
...
Ref.OpenVars(modName, r);
WHILE (r.mode # Ref.End) & (r.name # varName) DO
  r.Next
END;
IF r.form = Ref.Record THEN (*found*)
  ...
END
```

3. Applications

In this section we show how the Oberon-2 reflection model can be used to implement a number of interesting system services. In other programming systems, such services are often part of the programming language. Reflection, however, allows us to implement them outside the language in separate library modules so that they don't increase the size and complexity of the compiler. If a user does not need a service, he does not have to pay for it. Reflection also gives the system programmer a chance to adapt these services to special needs.

3.1 A Generic Output Function

In most modern programming languages input/output is not part of the language but is implemented in form of library functions. The problem with this approach is that it requires a separate function for every data type that is to be read or written. A typical output sequence could look as follows:

```
Out.String("Point (");
Out.Int(p.x);
Out.String(", ");
Out.Int(p.y);
Out.String(") is inside the search area");
```

Function overloading alleviates this problem, but still requires multiple function calls to print the above message. It would be nice to have a single generic function which is able to print any sequence of variables of a program with a single call. Reflection allows us to do that.

The following output function takes a string argument with the names of the variables to be printed. For example, the call

```
Put.S("Point (#p.x, #p.y) is inside the search area")
```

prints the argument string, but before printing it, it replaces every variable that is preceded by a # with the value of this variable.

The function *Put.S* is implemented with module *Ref*. It looks up the marked variables of the argument string in various scopes. For example, a variable *rec.arr[i]* is looked up as follows:

- *rec* is first searched in the local scope of the currently active procedure (using *OpenStack*) and—if not found—in the global scope of the current module (using *OpenVars*).
- If *rec* is found and turns out to be a record variable, a rider *r* is positioned on *rec*. *Put.S* zooms into *rec* (using *r.Zoom(r)*) and looks for a field *arr*. If it is found, the rider *r* is positioned on it.
- Since *arr* turns out to be an array, *Put.S* zooms into this array. It starts a new search for the variable *i* (using first *OpenStack*, then *OpenVars* as above). The value of *i* is read, the rider is positioned on the *i*-th element of *arr*, and the value of this element is read. This is the value of *rec.arr[i]*. It is inserted into the output string.

3.2 An Object Inspector

An object inspector is a debugging tool that can be used to inspect the values of the object fields. It can be conveniently implemented with Module *Ref*. To inspect an object that is referenced by a pointer *p*, one opens a rider *r* on the object's fields using *Ref.OpenPtr(p, r)* and iterates over them. Fields with a basic data type are simply shown with their values, whereas structured variables (arrays and records) are first represented in a collapsed form that that can be expanded on demand. When a collapsed variable is expanded, a new rider is placed on the inner elements (using *r.Zoom(r1)*) and is used to traverse the inner sequence.

Fig. 3 shows an example of a variable *head* that points to a list of three nodes. The middle part of the picture shows the list in collapsed state, the right part in expanded state. The triangles are so-called *active text elements* [MöK96] that hide the inner structure of an object. If the user clicks on a filled triangle, the object structure is expanded and shown between hollow triangles. A click on the hollow triangles collapses the structure again.

TYPE
  Node = POINTER TO
    RECORD
      value: INTEGER;
      next: Node
    END;
VAR head: Node;
	head = ^ ▶◀	head = ^ ▷
		  value = 6
		  next = ^ ▷
		    value = 5
		    next = ^ ▷
		      value = 4
		        next = NIL◁◁◁

Fig. 3. Object inspector view (collapsed and expanded)

A textual view like the one in Fig.3 is sufficient to represent data structures such as linear lists or trees, but it is less suitable for circular lists or graphs. For such purposes we have implemented a graphical tool that can also show circular data structures. This tool uses the same reflection mechanism as explained above.

3.3 Embedded SQL

SQL (structured query language) is a widely used standard for a database query language. It is normally used interactively from a dialog window. If a programmer wants to issue an SQL query from within a programming language (e.g. C++), however, he has to use an extended form of the language. For C++ there exists such an extension which is called *embedded SQL* [ESQL89]. It adds database query statements that are translated into library calls by a preprocessor.

We used a different implementation for embedded SQL that does not need any language extensions but is based on reflection [Ste96]. SQL queries can be specified as strings, which are passed to a function *Prepare* that analyzes and prepares them for execution. For example, one could write

```
stat := conn.Prepare("CREATE TABLE Persons FOR Person")
```

A prepared SQL statement can be executed with *stat.Execute*. Thus the statement can be executed several times without rebuilding internal data structures every time.

The SQL query can contain names of variables or types, which are then looked up in the calling program and are converted to appropriate data structures of the SQL libraries. For example, *Person* could be a type declared as follows:

```
TYPE
  Person = RECORD
    firstName, lastName: ARRAY 32 OF CHAR;
    age: INTEGER
  END ;
```

Its structure is used by the above SQL statement to create a table with the columns *firstName, lastName* and *age*. To distinguish program variables from database names (e.g. for tables and columns), variables are preceded by a colon in a query. In the following code fragement

```
VAR
  minAge: INTEGER;
  name: ARRAY 32 OF CHAR;
...
stat := conn.Prepare("SELECT firstName FROM Persons
WHERE age > :minAge INTO :name");
```

minAge and *name* are Oberon variables. They are looked up by reflection. The value of *minAge* is used to evaluate the WHERE clause. As a result, the database value *firstName* is transferred to the Oberon variable *name*.

Variables can be either scalar or of a record type. When record variables are specified, they are implicitly expanded to their fields. The statement

```
"SELECT * FROM Persons INTO :person"
```

is therefore equivalent to

```
"SELECT * FROM Persons INTO :person.firstName,
:person.lastName, :person.age".
```

3.4 An Exception Handling Mechanism

Exception handling is often part of a programming language, but it can also be implemented outside of the language, i.e. in a library module [Mil88]. Library-based exception handling is often implemented with the Unix functions *setjmp* and *longjmp*, which save and restore the machine state. We have followed a different approach based on reflection [HMP97]. Our technique has the advantage that it does not slow down programs as long as they do not raise exceptions.

Our exception handling mechanism is based on three concepts: a *guarded block* of statements which is protected against exceptions, one or more *exception handlers*, and a mechanism to *raise* exceptions (Fig. 4).

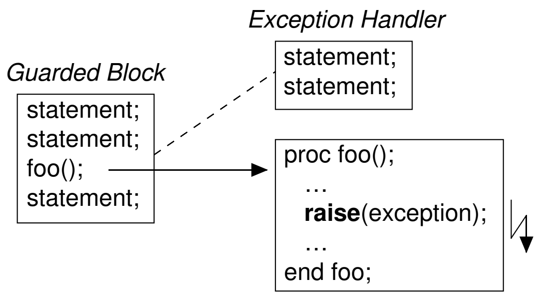


Fig. 4. Exception handling concepts

If an exception is raised in the guarded block or in one of the functions called from it, a suitable exception handler is called. After executing the handler, the program continues with the first statement after the guarded block. Exceptions are classes derived from a common exception class.

In our implementation, the concepts of Fig. 4 are mapped to the Oberon-2 language as follows:

- The guarded block is represented by an arbitrary procedure *P*.
- An exception handler is represented by a local procedure of *P*. It must have a single parameter whose type is a subclass of *Exceptions.Exception*.
- An exception is raised by the call of the library procedure *Exceptions.Raise(e)* where *e* is an object of an exception class.

The following code fragment shows an example (The classes *Overflow* and *Underflow* are subclasses of *Exceptions.Exception*).

```
PROCEDURE GuardedBlock;
  VAR ofl: Overflow; ufl: Underflow;

  PROCEDURE HandleOfl (VAR e: Overflow); …
  END HandleOfl;

  PROCEDURE HandleUfl (VAR e: Underflow); …
  END HandleUfl;

BEGIN
  …
  IF … overflow … THEN
    … fill the ofl object with error information …
    Exceptions.Raise(ofl)
  END;
  …
END GuardedBlock;
```

In this example, *GuardedBlock* raises an exception by calling *Exceptions.Raise(ofl)*. Procedure *Raise* is implemented with reflection. It determines the type of *ofl* and searches through the local procedures of *GuardedBlock* to find a procedure with a matching parameter type. This is the exception handler (in this example *HandleOfl*). If such a handler is found, it is called. Finally, the activation stack is unrolled so that the control is returned to the caller of *GuardedBlock*.

If no matching exception handler is found in *GuardedBlock*, the lookup continues in the caller of *GuardedBlock*. If this caller *P* contains a local procedure *H* with a matching parameter type, *H* is called as an exception handler, and then the program continues with the first statement after *P*.

If no exception handler is found in any of the currently active procedures, the program is aborted with a standard error message. The following pseudocode fragment shows the implementation of *Raise*:

```
PROCEDURE Raise (VAR e: Exceptions.Exception);
  E := dynamic type of e;
  FOR all stack frames in reverse order DO
    P := procedure of this frame;
    FOR all local procedures H of P DO
      IF H has a parameter of type E or a subtype THEN
        Invoke H(e);
        Return to the caller of P
      END
    END
  END
END Raise;
```

Except of the underlined parts, all actions of *Raise* are implemented with the reflection model described in Section 2. In particular, riders are used to traverse the stack frames, the procedures and the parameters to perform the handler lookup. The underlined actions involve stack manipulation. They are implemented using low-level facilities of Oberon-2 and are not show here (see [HMP97]). The dynamic type of an object can be obtained with an Oberon system function.

The exception handling mechanism is encapsulated in a library module with the following simple interface:

```
DEFINITION Exceptions;
  TYPE Exception = RECORD END; (*abstract base class*)
  PROCEDURE Raise (VAR e: Exception);
END Exceptions.
```

4. Discussion

In this section we discuss some consequences and tradeoffs of the Oberon reflection model, namely security and performance issues as well as the interference of reflection with optimizing compilers.

4.1 Security

The Oberon reflection model gives the systems programmer full access to all variables, types and procedures in a program, even to the private objects that are not exported from a module. This of course raises the question of security. The visibility rules of the language can be circumvented by reflection, however, this is necessary to implement system software such as debuggers or inspectors.

Although the whole structure of a program is visible to reflection, it is not possible to access it in an undisciplined way. The Oberon reflection model is strongly typed. All metainformation is read and written according to their types. It is never possible, for example, to access a pointer as an integer or vice versa.

The most critical fact about the Oberon reflection model is that data can not only be read but also written. This is sometimes necessary as for example in the Embedded SQL facility described in Section 3.3. One should use this feature very carefully. According to our experience most reflective applications don't make use of it.

Clearly, reflection allows a programmer to do more than what he could do with an ordinary programming language. But this is exactly its advantage. System programmers need sharper tools than application programmers.

4.2 Interference with Optimizing Compilers

An optimizing compiler may decide to keep variables in registers rather than in memory, to eliminate variables at all, or to introduce auxiliary variables, which are not in the source program. Some of these optimizations are easy to cope with in the reflection model, others are more difficult to handle. The Oberon-2 compiler does not perform aggressive optimizations. In addition to some code modifications (which do not affect the metainformation) the compiler keeps certain variables in registers. For such variables, their register numbers are stored in the metainformation, so that riders can find them. The Oberon-2 compiler does not eliminate, introduce or reorder variables. But even such cases could be handled if the metainformation carried enough information about the optimizations that the compiler performed.

4.3 Performance

The Oberon reflection model leads to small memory overhead at the cost of run time efficiency. The layout of our metainformation is sequential. For example, in order to access the information of the third local variable in the fourth procedure of a module, one has to skip three procedures, zoom into the fourth one and search for the third variable. Table 1 shows the approximate costs for reading (i.e. skipping) various kinds of elements in the metainformation sequence. Of course, the time to skip a procedure or a record type depends on its size.

Table 1. Access times for specific program elements (on a Pentium II with 300 MHz)

Element to be skipped	*Cost*
Local variable	2 µs
Global variable	2 µs
Record field	2 µs
Record type	9 µs
Procedure	15 µs

The sequential layout of the metainformation as described in Section 2.1 requires us to skip all record types before we get to the first procedure. Furthermore, in our current implementation, the global variables of a module are treated like local variables of the module body, which is considered to be a special procedure. Accessing them requires us to skip to this procedure first. Constraints like these make random access of meta-information elements somewhat inefficient. In practice, however, a typical access pattern involves both random access and sequential access so that the run time was never a serious problem in all the applications described in Section 3. As a task for further research one could try to redesign the metainformation so that it is indexed and random access becomes more efficient.

Our current implementation has the advantage that the metainformation is stored in a very compact form. For example, the metainformation of the whole Oberon compiler (9 modules, 413 procedures, 14 types, 1690 variables and 86 record fields) consumes only 23706 bytes on the disk. When it is accessed it is cached in memory so that it is not necessary to go to the disk for every access. Reading the meta-information of the whole compiler sequentially takes 30 milliseconds.

5. Summary

The fundamental difference between the Oberon-2 reflection model and other models is that the metainformation is not accessed via metaobjects. It is rather parsed on demand from a file (which is usually cached in main memory to avoid file operations). One advantage of this approach is the reduced number of objects needed to represent the metainformation and the reduced memory consumption. For example, when meta-information is used in a post-mortem debugger to produce a readable stack dump, it is important not to waste memory since the reason for the trap might be the lack of memory. A disadvantage of our approach is that the information is parsed again and again. But this is not time-critical as the measurements show.

The Oberon-2 reflection model is currently designed for convenient access to the structure and values of program objects. In the future, we plan to extend it with mechanisms for calling and intercepting methods.

Acknowledgements

The Oberon-2 reflection model was originally designed by Josef Templ [Tem94] in his PhD thesis. We refined his model and adapted it for our purposes. Our colleagues Markus Knasmüller [Kna97], Markus Hof [Hof97], Martin Rammerstorfer and Günter Obiltschnig [Obi98] used the model to implement various applications on top of it. We wish to thank the anonymous referees for their comments that helped us to make the paper more focussed.

References

[ArG96] Arnold K., Gosling J.: The Java Programming Language. Addison-Wesley, 1996

[Att89] Attardi G., et al.: Metalevel Programming in CLOS. Proceedings of the ECOOP'89 Conference. Cambridge University Press, 1989

[ESQL89] Database Language – Embedded SQL (X3.168-1989). American National Standards Institute, Technical Committee X3H2

[GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[HMP97] Hof M., Mössenböck H., Pirkelbauer P.: Zero-Overhead Exception Handling Using Metaprogramming. Proceeding of SOFSEM'97, Lecture Notes in Computer Science 1338, 1997

[Hof97] Hof M.: Just-In-Time Stub Generation, Proc. Joint Modular Languages Conference '97, Hagenberg, Lecture Notes in Computer Science 1204, Springer-Verlag, 1997

[Kna97] Knasmüller M.: Oberon-D, On Adding Database Functionality to an Object-Oriented Development Environment, Dissertation, University of Linz, 1997

[LMN93] Lehrmann-Madsen O., Moller-Pedersen B., Nygaard K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, 1993

[Mil88] Miller W.M.: Exception Handling without Language Extensions. Proceedings of the USENIX C++ conference, Denver CO, October 1988

[MöK96] Mössenböck H., Koskimies K.: Active Text for Structuring and Understanding Source Code SOFTWARE - Practice and Experience, 26(7): 833-850, July 1996

[MöW91] Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991

[Obi98] Obiltschnig G.: An Object-Oriented Interpreter Framework for the Oberon-2 Programming Language, Diploma Thesis, University Linz, 1998

[Ste96] Steindl Ch.: Accesing ODBC Databases from Oberon Programs. Report 9, University of Linz, Institute for Practical Computer Science, 1996

[StM96] Steindl Ch., Mössenböck H.: Metaprogramming facilities in Oberon for Windows and Power Macintosh. Report 8, University of Linz, Institute for Practical Computer Science, 1996

[Tem94] Templ J.: Metaprogramming in Oberon. Dissertation, ETH Zurich, 1994

[WiG89] Wirth N., Gutknecht J.: The Oberon System. Software—Practice and Experience, 19(9):857-893, 1989