Structuring a Compiler with Active Objects

Patrik Reali

Institut für Computersysteme
ETH Zürich
reali@inf.ethz.ch

Abstract. We present a concurrent compiler for Active Oberon built itself with active objects. We describe the experience made on parallelizing the Oberon compiler, in particular explaining how concurrency and synchronization are achieved by using active objects, and showing how we achieved ensured deadlock freedom. Based on the implementation, we discuss why we consider active objects beneficial for the implementation of software frameworks and where their limitations are.

1 Introduction

It is almost a tradition to write a compiler in the language it compiles. There are mainly two reasons for this: a compiler is usually a program complex enough to challenge a language and possibly find weaknesses to be removed. The other reason is that in the long range one wants to use only one programming language in a system.

In this paper we show the implementation of a compiler for the Active Oberon language written in Active Oberon. The compiler makes essential use of the concurrency facilities of language. Since active objects are a new concept in the language, we want to use them to gain experience and take advantage of the increased implementation and design potentials they offer. The recent availability of symmetric multiprocessor (SMP) machines at affordable prices is also a motivation to investigate the use of concurrency constructs in the language.

In Section 2 the main features of the language are shortly recapitulated; section 3 introduces the field of concurrent compilation and the previous projects that inspired this work; section 4 discusses the implementation details of the compiler, in particular the concurrent parser and the symbol table, which serves as a common shared data structure; section 5 discusses a new problem introduced in the compiler by the parallelism: deadlock; section 7 then draws the conclusions of this work.

2 Active Objects in Oberon

This section recapitulates the Active Oberon language [7], an extension of the Oberon language [16, 18]; it is a homogeneous integrated concurrent programming language that embeds concurrency support in Oberon, by making just

minimal additions to the language and creating a unified framework for concurrent, object-oriented programming. This is done by introducing concurrent activities, protected access and guarded assertions in the language. Figure 1 give an example of an active object.

```
TYPE
  MovieProjector* = POINTER TO RECORD
    frame: ARRAY N OF Frame;  in, out: INTEGER;  window: Window;

    PROCEDURE GetFrame*(): Frame;
    VAR  f: Frame;
    BEGIN {EXCLUSIVE}
      PASSIVATE(in # out);
      f := frame[out];  out := (out+1) MOD N;
      RETURN f
    END GetFrame;

    PROCEDURE PutFrame*(f: Frame);
    BEGIN {EXCLUSIVE}
      PASSIVATE((in+1) MOD N # out);
      frame[in] := f;  in := (in+1) MOD N
    END PutFrame;

    PROCEDURE & Init(window: Window);
    BEGIN
      SELF.window := window;  in := 0;  out := 0
    END Init;

  BEGIN
    LOOP
      time := Timer.Time();
      DrawFrame(window, GetFrame());
      Timer.WaitUntil(time + FrameDelay)
    END
  END MovieProjector;
```

Fig. 1. Example of an Active Object

Compared to Oberon, record types have been upgraded in Active Oberon to have methods (also called type-bound procedures) and an own body. Syntactically, this is done by generalizing the scope concept and applying it to records as well. The methods have privileged access to the record fields, while the body (to be considered itself a method) represents the object's own activity, started asynchronously at the time of object instantiation. The object's activity corresponds to a thread of execution (the two terms will be used here as synonyms).

To protect the objects internal data structure against concurrent access, every object instance acts as a monitor. Methods using the `EXCLUSIVE` clause make use of protection by ensuring mutual exclusion in the sense that at most one protected method is allowed to be active in an object instance. A module is considered a singleton object instance, its procedures may also be protected.

Any thread of execution can be synchronized with a system or object state by using guarded boolean assertions in a `PASSIVATE` statement; the thread is preempted until the guarded condition becomes true. Such a thread is said to be *passivated*. It is noteworthy that the operating system itself is in charge of evaluating the conditions and eventually reschedule the passivated threads. When a thread is passivated in an object, it releases its mutual exclusion on that object as long as it remains passivated.

Some other minor adjustments in the language have been made to smoothly integrate the previous changes. First, a symbol named `SELF` is implicitly defined in every procedure scope, it refers to the current object the procedure is in. Second, it is possible to extend a pointer-based record (a structure with only dynamic instances), but the extending type must also allow only dynamic instances. Third, forward declarations are no longer needed, because they express only redundant information.

Our current system [10] has some implementation restrictions that had to be considered during this work: 1) reentrant locks are not allowed; 2) shared locks are not allowed; 3) global conditions to be passivated on are not allowed; 4) active modules are not allowed. We are conviced that these features should be removed from the language, because they are either redundant or conceptually wrong.

3 Building a Concurrent Compiler

3.1 Classical Compiler Architecture

There is a big experience in building compilers and many textbooks explain this task in detail [15, 17, 1] (just to cite some). Compilation is divided in phases like lexical analysis, parsing, semantical analysis, data allocation, register allocation, optimization and code emission that may be grouped into passes. The number of phases may vary, depending on the language parsed and the quality and expected complexity of the generated code. The number of passes is a strategic decision in building the compiler; merging all the phases in only one pass can make the compiler very fast, but on the other hand not every language can be parsed in one pass; also many optimizations require a separate pass. In general, more phases allow for more flexibility, but also increase the program complexity [9, 15, 8, 14].

Many compilers have been developed at ETH under the supervision or inspiration of N. Wirth [17]. All share the common goal of simplicity, compactness, efficiency and the use of single pass top-down parsing. OP2 [2], currently used in many ETH-Oberon systems, is an evolution that divides the compiler in a

front-end and a back-end with a shared syntax tree to allow easy re-targeting and more aggressive code optimizations. We based our work on these previous implementations.

3.2 Previous Work

The first multiprocessor machines in the 1970 already challenged the compiler makers to investigate concurrent compilation.

C. Fischer [3] and R. Schell [12] where among the first to investigate the theoretical feasibility of concurrent compilation, by seeking for concurrent parsing techniques. Two approaches are possible: dividing compilation into concurrent phases that execute in a pipelined way or dividing the source code into pieces that are compiled concurrently.

While phase pipelining is limited in its speedup, because the compiler cannot be made faster than the slowest phase, splitting the code in concurrently compiled pieces can result in a good speedup [6, 19].

The Toronto Modula-2+ compiler project [19] investigated the implementation of a parallel Modula-2+ compiler on a multiprocessor machine; in that compiler every scope is compiled concurrently and the symbol table is used to synchronize the concurrent parsers. Many of the results can be applied to our compiler on account of the similarities between Active Oberon and Modula-2+.

One obvious problem found in the Toronto compiler is the *doesn't know yet* problem (DKY). It is possible that a visible symbol exists in a scope but has not been processed by its parser and thus is not visible yet. In this case a parser requiring it *doesn't know yet* whether or not a valid declarations exists in the incomplete scope.

Seshadri et al. [13] propose two strategies to handle the problem. *DKY Avoidance* approaches the problem by ensuring that the scopes are complete before their symbols are used; *DKY Handling* treats it by inserting fix-ups to be patched at a later time or by making assumptions on the symbol. They assert that simplest solution is to use DKY Handling in the declaration part of a scope and DKY Avoidance in the implementation part. We will pursue this strategy in our compiler: a parser will be passivated until the desired symbol has been processed by another parser.

4 Implementation of the Compiler

4.1 Parser

The top-down recursive descent paring technology is used. Parsers can be very easily generated by systematically translating the productions into procedures.

Concurrent parsing is implemented by a generic active parser object whose activity is in charge of parsing the definition and implementation parts of a scope, the `DeclSeq` and `StatSeq` productions of the language. Figure 2 shows the generic Active Parser.

```
TYPE
  Parser = POINTER TO RECORD
    VAR  scope: Scope;  scanner: Scanner;
    ....
  BEGIN  (*active body*)
    DeclSeq;
    IF sym = begin THEN  scanner.Get(sym);  StatSeq  END
  END Parser;
```

Fig. 2. The generic Active Parser

For parsing modules, procedures and records we define extensions of the generic parser each, in charge of initializing the parser depending on the specific properties of the scope and positioning the scanner at the begin of the declaration section. Figure 3 shows the declaration of the procedure parser object: it prepares itself for the procedure parsing, while the rest of the scope is parsed by the inherited generic parser, which is the same for all the scopes.

```
TYPE
  ProcedureParser = POINTER TO RECORD (Parser)
    PROCEDURE & InitProcedure(scope: Scope;  scanner: Scanner);
    BEGIN SELF.scope := scope;  SELF.scanner := ForkScanner(scanner)
    END InitProcedure
  END ProcedureParser;
```

Fig. 3. The Module Parser

Every time a record or a procedure is encountered by a running, a new parser for it is generated and the scope contents are ignored by the running parser, delegating the parsing to the newly generated parser. Figure 4 shows the parsing of a procedure. Instead of calling directly `DeclSeq` and `StatSeq` a new parser is allocated.

4.2 Symbol Table

The symbol table uses three type hierarchies: `Object`, `Struct` and `Scope`. `Object` models symbols in the table such as variables and procedures; `Struct` models structures such as basic types and composed data types, `Scope` contains a list of symbols and methods to manipulate it.

The symbol table is shared among all the concurrent parsers. It must be protected against concurrent access. We want to hide the concurrency details in the symbol table as much as possible, to minimize the knowledge (and so the troubles) that the synchronization requires.

```
PROCEDURE ProcDecl;
VAR  parser: ProcedureParser;  procscope: ProcedureScope;
BEGIN
  NEW(procscope);
  CheckToken(Procedure); IdentDef(procscope.name, mark);
  FormalParameters;  CheckToken(semicolon);
  NEW(parser, procscope, scanner);
  SkipScope;
END ProcDecl;
```

Fig. 4. The ProcDecl production implementation

Scopes are used like state machines. There are three relevant scope states: *filling* means that the scope is not complete and more symbols may still be added; *checking* that all the symbols have been inserted, and are being semantically checked; *complete* that the scope is ready to be used. The way scope operations work, depends on the state a scope is in.

Figure 5 shows the implementation of the `Scope` type. Every scope is owned by a parser; the owner is the only parser allowed to change the state of the scope.

`Insert` appends a symbol to the list. The owner is allowed to insert symbols only while the scope is in the *filling* state.

`Find` is in charge of handling the DKY problem in a way, that hides the details from the parser. In the *filling* state only the owner is allowed to access the scope, because it needs it for completing the definitions; if a symbol cannot be found in the current scope, a fix-up is created which will be resolved when the scope is complete: at this point it is still not possible to tell if the symbol is local to the scope or belongs to a parent scope or is not defined at all. Queries from other parsers are passivated as long as the scope is not complete.

The given implementation already carries out DKY Handling for declarations and DKY avoidance for implementation. The *filling* state correspond to the parsing of the implementations for the current scope: only the current parser is allowed to search for symbols in the scope; if no symbol is found, a fix-up is created because the scope doesn't know whether the symbol will be declared in the same scope or not; the search doesn't need to be propagated to the parent scope (yet) and the fix-up will be patched later (DKY handling). Other parsers, usually in the child scopes, may have to lookup symbols in the current scope too; they are preempted until the scope is complete, thus avoiding the handling of incomplete declarations (DKY avoidance). Note that only parsers in the implementation section are allowed to lookup symbols in parent scopes. A parser should not perform any semantic check or operations on the symbols during the *filling* state, because they may be a fix-up, thus carrying no information.

The `Insert` and `Find` procedures do not need access protection but only synchronization with the scope state, because the design ensures that only one parser will change the scope, and during that time (state = *filling*) no other parser is allowed to access the scope. Only the procedures to change a scope's

```
TYPE
  Scope* = POINTER TO RECORD
    VAR  parent: Scope; list: Object; state*: State;
      ....
      PROCEDURE Insert*(p: Object);
      BEGIN
        ASSERT(CalledByOwner() & (state = filling));
        Append(list, p)
      END Insert;

      PROCEDURE Find*(name: ARRAY OF CHAR): Object;
      VAR  p: Object;
      BEGIN
        (* code: DKY Avoidance *)
        IF ~CalledByOwner() THEN PASSIVATE(state = complete) END;

        p := Lookup(list, name);

        (* declarations: DKY Handling *)
        IF (p = NIL) & (state = filling) THEN p := <do fix-up> END;

        IF (p = NIL) & (parent # NIL) THEN p := parent.Find(name) END;
        RETURN p
      END Find;
  END Scope;
```

Fig. 5. The `Scope` type

state need to be protected. This is similar to a simplified readers-writers schema; once the only writer has terminated the writers are free to access the scope and no reader is allowed anymore.

Active Oberon removes forward declarations, hence forward references of symbols must be handled by the compiler. Instead of using a 2-pass parsing algorithm, we choose to take advantage of the concurrent parsers. Forward declarations are just a special case of DKY problem: they are symbols whose declaration come textually later than their use. As the compiler is already prepared to handle DKY cases, this is just one restriction less on the parsing, namely that declarations must come before their use.

4.3 Symbol Checking

The DKY handling for declarations is still not complete: when shall fix-ups be resolved? Only when the scope is complete, the semantic check of its symbols and related structures can be done. We show two approaches to solve this problem.

A first approach is to automatically check all the symbols in the scope after the declarations are completely parsed. A second approach is to do the fixing and checking on demand, when the symbol is first required. The *Find* procedure in the scope can intercept the access to an unchecked symbol and start its checking, which can trigger the checking of the associated structure, and return only after the symbol is completely checked.

To ensure the table consistency, access to the data must be restricted during the check operation. At first sight four lock granularities are possible: the whole symbol table, the scope, the whole hierarchy of the structure or the structure to be checked. Since using the first variant does check all the objects in a scope, it is appropriate to use the scope as lock granularity and forbid access to it during the whole operation. The implementation of the second variant (check on request) can get very complex, depending on the granularity chosen: locking single objects is not enough, since in case of circular structures (with pointers or ill-declared ones) two parsers may start checking two different parts of the structure hierarchy and then run into each other's locks creating a deadlock situation; locking the whole table is too coarse since only the current scope and its parents are visible and thus may be used for the check; locking the scope is exactly what is done in the first variant, with the disadvantage that later the scope should still be traversed for unchecked symbols[1].

Conceptually, the most appropriate lock granularity would be a type hierarchy (e.g. a type and its base types). A first problem is that we don't know a type hierarchy a priori, because we first have to check the single types of the hierarchy; we also don't have an object modeling a type hierarchy nor it is possible to model it (especially in the case of ill-defined structures[2]). Locking would be thus

[1] the fact that a symbol is never used doesn't make its declaration correct

[2] Somebody could object that we don't have to accept ill-defined types; to be able to detect them, the types must first be checked and thus locked. This is a chicken-egg problem.

done by locking the single structures, but this is cannot be done as explained before.

We thus choose the first variant on account of its simplicity compared to the minimal efficiency improvement of the second variant at the cost of a much more complicated implementation.

A more subtle problem is caused by the built-in types of Oberon. These types are not defined as keywords; this allows the programmer to redefine their names with a new meanings. The consequence is evident: a parser must traverse all the parent scopes to decide if an `INTEGER` is the built-in integer type or has a user defined meaning. Since those types are the most heavily used ones and Oberon supports nesting of scopes, this can cause an efficiency problem. The parallel parsing doesn't help either, since a parser can't assume that the parent scopes have already been checked; it may have to wait for their conclusion before being able to access the build-in types, incurring in an even longer delay. We thus implemented the built-in type's names as reserved words and hence do not allow their redefinition. This doesn't change the expressiveness of the language, because type aliasing is always possible. This did not cause any problem, since no single module in the whole Native Oberon release does redefine a basic type.

5 Deadlock Avoidance

Using concurrent processes can create deadlock situations: the parsers may depend on each other for the parsing and thus for making the symbols in their scope available. Whenever a circular dependency between parsers exists, deadlock cannot be excluded.

As described in the previous sections, every scope goes through three stages: filling, checking, complete. The only synchronization in the whole compiler is done in the symbol table, in the `Find` procedure, thus this is the only place where a deadlock situation may occur. The following two rules describe how Find (see Figure 5) works: 1) lookups during the filling and checking phase are allowed only for the parser owning the scope, other parsers lookups' are passivated until the scope reaches the complete state; 2) lookups during the filling phase are local to the scope, otherwise they may be propagated to the parent scopes.

We assume that every parser will always lookup symbols in the own or in a parent scope, never in a child one. To make this assumption hold, we aggregate (conceptually is enough) the interface information of every symbol to the parent scope where the symbol itself is declared. For procedures this means the formal parameters, for records all the fields and methods[3]

We first analyze the case of the parser for the top scope (the one that has no parent scope): in the filling state, Find always terminates, because a symbol is either found or a fix-up for it is created and returned; in the checking and complete states the lookup also always terminates: being the owner of the scope, the thread doesn't get preempted and being the top scope, the search is

[3] In Oberon, record fields and methods are always visible in the whole module

not propagated to a parent scope, thus always terminating. Since Find always terminates, the parser can process the declarations, check them and eventually reach the complete state.

In the generic case, a scope has a finite number of parent scopes. In the filling state, the lookup always terminates as in the top scope case; in the checking and complete state, when the object is not locally declared, the lookup may have to be done in the parent scopes. Since the top scope eventually reaches the complete state, its children will be allowed to search through it, hence allowing them to eventually reach the complete state themselves. Of course, a scope using only local symbols will reach the complete state without having to wait for the parent scope to be completed. It is possible to inductively show that every scope in the hierarchy will eventually reach the complete state, and thus that the search operation terminates.

6 Discussion

6.1 Data Structures in the Compiler

Using type extension as proposed in [5, 4] instead of records with many different meanings made the compiler not only safer but also simpler to understand and maintain. Invariants are hard-coded in the data structure, and inconsistent structures can mostly be detected when created and not when first used. We thus go one step further in using the language facilities to have invariants automatically tested. Using all the language facilities to ensure program correctness is very important and saves time with debugging. We can only agree with E. Raymond's quote [11] (paraphrasing F. Brooks): "Smart data structures and dumb code works a lot better than the other way around".

On the other hand, having a separate type for every kind of object makes the source code longer, while the program becomes slightly shorter, because less invariants must be explicitly checked. It's noteworthy that the data structures don't affect the size of the compiled code and that memory usage is reduced because the structures are smaller. This also simplifies program documentation, because the data structures are more expressive and don't need to be interpreted in terms of a mode field.

We also planned to protect our structures with information hiding, but finally we did not. To execute efficiently, a compiler needs all the possible informations available: hiding information would make the compiler slower, because the same information would have to be recomputed many times or accessed through procedure calls. On the other hand we realized that there is no need to access the information through methods: the compiler usually adds information, but never changes it, thus the data must be only checked when created. Read-only export of the data in this case would be enough to ensure that the invariant, once established, will never be broken, while giving a fast and efficient access to the data. Hence we think that a read-only export option is a necessary addition to the language.

6.2 Concurrency in the Compiler

We used an active object to model a generic concurrent parser. Making parsing parallel was rather simple and required just to encapsulate all the parsing procedures and the global variables in the parser object, thus making them a state of every single parser. Our concurrent parser is very similar to the usual 1-pass parser, but forks a new parser when a scope is encountered. The synchronization is almost completely hidden in the symbol table, thus very local and simple to understand. Nevertheless it makes the handling of the symbol table more complex. Deadlock freedom becomes a new complex issue in the compiler.

Concurrent parsing elegantly allows to handle forward references in the implementation section by using synchronization instead of a second parsing pass. It is rather interesting that this is done by removing an artificial sequentialization, thus in fact by removing a restriction in the code[4] and an anomaly in the scope rules: in Oberon an identifier can have two meanings in the same scope, depending on the position it is defined:

[4] the check that a declaration comes before its use

```
TYPE  BaseDesc = RECORD .... END;
PROCEDURE P;
  TYPE
    Base1 = POINTER TO BaseDesc;
    BaseDesc = ARRAY 5 OF INTEGER;
    Base2 = POINTER TO BaseDesc;
    ....
END P
```

In the previous piece of code, `BaseDesc` has two meanings in procedure P. This contrast with of the Oberon language report: *"No identifier may denote more than one object within a given scope (i.e. no identifier may be declared twice in a block"* (§4.1). By making the visibility only scope-based, `BaseDesc` becomes unequivocal in the scope of P.

6.3 Using Active Objects

In our work, we found the protection of objects amazingly simple to use and understand. It is very helpful to declare whole procedures as exclusive and it makes the code readable, because the protected regions can be clearly identified.

Using active objects is simple, as long as the granularity needed to solve the problem is the same as the granularity of the objects; protection and synchronization become straightforward. It is thus convenient to consider the protection granularity when modeling the data structures. In the compiler, the semantic check of types in the symbol table (sec. 4.3) is a good example: granularities ensuring a correct program are table, scope and type-hierarchy granularity. A finer granularity has less collision chances and is thus more efficient. In our design is not possible to lock a type-hierarchy, because we don't have a data structure modeling it, we thus use the scope granularity.

The use of guarded assertions (`PASSIVATE`) is a step toward better abstract design facilities. A conventional implementation using signals has to disclose its internal state to other threads in order to allow them to decide if and when they should send the signal. Of course the other threads have to know that a signal has to be sent and have to be willing to do the favor. This again is an invariant enforced solely in the documentation! Especially when using software frameworks, it is easy to forget these invariants. Using guarded assertions this potential source of errors is removed, because the system is in charge of restarting passivated threads whenever the condition is established, thus enforcing the invariant automatically without requiring external intervention or even knowledge. We can thus assert that conditional guarded assertions allow an improved information hiding and safer synchronization for concurrent programs, and that the architecture scales up well in software frameworks.

Proving deadlock freedom of a program is still a burden left to the implementor and the only help that active objects can give, is by making the programs simpler to understand and thus simpler to reason about. In this project we used the hierarchical ordering of the threads to prove that the program is structurally deadlock free.

7 Conclusions

In this paper we reported the experiences made using active objects to implement a concurrent compiler. A concurrent parsing technique allows us to parse forward references by using a slightly modified recursive descent parser technology. The complexity of the synchronization is almost completely hidden in the symbol table.

With this project we challenged Active Oberon's usability. Active objects are a valuable tool that simplifies the implementation of concurrent programs. Data protection from concurrent access is remarkably simple when the locking granularity is the same as that of the designed structure; the synchronization of threads by means of guarded assertions is very elegant and safe. Frameworks can benefit from it, because it minimizes the needed knowledge of the whole framework and ensures a safe resumption of passivated threads without user intervention.

References

1. A. W. Appel. *Modern Compiler Implementation In Java, basic techniques.* Cambridge University Press, 1997.
2. R. Crelier. OP2: A portable oberon compiler (vergriffen). Technical Report 1990TR-125, Swiss Federal Institute of Technology, Zurich, February, 1990.
3. C. N. Fischer. *On parsing Context-Free Languages in Parallel Environments.* PhD thesis, Department of Computer Science, Cornell University, 1975.
4. P. Fröhlich. Projekt froderon: Zur weiteren entwicklung der programmiersprache oberon-2. Master's thesis, Fachhochschule München, 1997.
5. R. Griesemer. *A Programming Language for Vector Computers.* PhD thesis, ETH Zürich, 1993.
6. T. Gross, A. Zobel, and M. Zolg. Parallel compilation for a parallel machine. *ACM SIGPLAN Notices*, 24(7):91–100, July 1989.
7. J. Gutknecht. Do the fish really need remote control? A proposal for self-active objects in oberon. In *Proc. of Joint Modular Languages Conference (JMLC). LNCS 1024*, Linz, Austria, March 1997. Springer Verlag.
8. Jürg Gutknecht. One-pass compilation at its limits — A Modula-2 compiler for the Xerox Dragon computer. *Software Practice and Experience*, 17(7):469–484, July 1987.
9. D. E. Knuth. A history of writing compilers. *Computers and Automation*, 11(12):8–14, 1962.
10. P. Muller. A multiprocessor kernel for active object-based systems. In *Proc. of Joint Modular Languages Conference (JMLC2000)*, Zürich, Switzerland, September 2000. Springer Verlag.
11. Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates, Inc., 1999.
12. R. M. Schell, Jr. *Methods for Constructing Parallel Compilers For Use in a Multiprocessor Environment.* PhD thesis, University Of Illinois at Urbana-Champaign, 1979.
13. V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 233–240, 1988.

14. J. C. Sheperd. Why a two pass front end? *SIGPLAN Notices*, 26(3):88–94, March 1991.
15. Aho; Sethi; Ullman. *Compilers; Principles, Techniques and Tools*. Addison-Wesley, 1986.
16. N. Wirth. The programming language oberon. *Software Practice and Experience*, 18(7):671–690, July 1988.
17. N. Wirth. *Compiler Construction*. Addison-Wesley, 1996.
18. N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, first edition, 1992.
19. D. Wortman and M. Junkin. Concurrent compiler for Modula-2+. *ACM SIGPLAN Notices*, 27(7):68–81, 1992.