

The Case for Universal Symbol Files

Michael Franz

Institut für Computersysteme, ETH Zürich, CH-8092 Zürich, Switzerland

Abstract. Many compilers for modular languages preserve the interface information obtained during compilation on symbol files. This eliminates the need to read the source text again when the interface of a previously compiled module is imported by client modules. Instead, the compiler retrieves the interface information from the symbol file, which is usually much more efficient than processing source text.

Traditionally, the format of symbol files has been tied intimately to the inner workings of the symbol-table handler. We argue that the preconditions urging so close a relationship between these internal and external representations are no longer valid and present several methods for improving the symbol-table encoding. Combined, these methods yield concise symbol files that are completely independent of the architecture of the target machine. We also suggest how the notion of universality can be extended to module keys and present a simple version-stamping protocol based on fingerprints.

Keywords: compiler construction, separate compilation, module interfaces, software portability, programming languages, Oberon, symbol file

1. Introduction

Programming languages such as Mesa [8], Ada [11], Modula-2 [13], and Oberon [14] support the separate compilation of program *modules* (called *packages* in Ada). This means that the compiler performs full consistency checking across module boundaries, as if the source texts of all modules involved in a compilation were contained in one large program file. However, unlike simpler multifile-compilation mechanisms in which the text of some header files is included literally in the main program prior to compilation and the resulting multifile document is considered a single compilation unit, separate compilation keeps individual modules distinct. As a consequence, certain modifications of individual modules can be performed without affecting other modules.

Separate compilation works by allowing a module to *export* some features that may subsequently be *imported*

by other modules. All intermodule references are established by such import/export relationships, which are resolved in a process separate from compilation, called *binding* or *linking*. In modern systems, linking takes place at the time of loading and is therefore invisible to the user (*dynamic loading*). In the following, we will sometimes call an exporting module a *library*, while the modules that import from it are called its *clients*. Client modules may themselves serve as libraries to other clients further up in the module hierarchy, so that layer upon layer of a system can be built upon simpler abstractions below. However, we require that the module dependency graph be acyclic.

Naturally, separate compilation of a modular application is more complex than compiling a monolithic, self-sufficient program. In order to perform intermodule type and parameter checking, the compiler has to process the declarations imported from library modules in addition to the source text of the module being compiled. The simplest way to achieve this is by always recompiling the complete module hierarchy from the bottom upwards, retaining in memory the symbol tables of library modules for the compilation of their clients. Foster [1] describes a system in which all interface information is obtained from source text in this manner, to which effect the compiler invokes itself recursively while traversing the module dependency graph from its root (i.e., the “main module”).

However, extracting interface data from a source program is a costly operation. Repeating it for the compilation of every client module may slow down compilation considerably. It has therefore become an established practice to preserve the interface information obtained during compilation in an efficiently readable, coded intermediate format, so that it need not be reconstructed from source text for every compilation of a client module. A file containing such a coded interface description is called a *symbol file* [4, 6].

Although symbol files were invented as a means of increasing compilation speed, they are useful for another reason that is more organizational than technical. A

symbol file represents a frozen state of a certain interface, while it is not directly editable by humans. This helps to discipline programmers, restraining them from changing interfaces too light-heartedly. When larger software systems are being constructed, the interfaces between self-contained subsystems are often specified early in the design cycle and compiled into symbol files. The different parts of the system can then be implemented (and later maintained) concurrently by separate teams of programmers. If the original set of symbol files is used in every compilation, one can be sure that the parts will fit together without exception.

In combination with version stamping, the use of symbol files can also simplify the consistency checks that are required at the time of linking to determine whether libraries and client modules are compatible with each other. This particular use of symbol files will be discussed in the last section of this paper.

2. The Path to Universal Symbol Files

In the programming language Oberon [14], there is a single source text for every program module. This document specifies a module's interface as well as its implementation and is used by the compiler to generate both a symbol file and an object file. In some implementations, these two are actually only parts of a single physical file. After compilation, the newly generated symbol file represents the module's interface and is subsequently processed instead of the source text when client modules are compiled.

Each symbol file contains essentially a linearized copy of the compiler's symbol table, or, more precisely, a subset of this table describing the features that are visible outside the module being compiled, plus some additional features that are considered invisible but required for achieving completeness. For example, the export of a variable forces the inclusion of its type in the symbol file, although the type itself may not be exported explicitly. In this case, the type needs to be flagged as invisible, telling the compiler that it should forbid programmers to declare further variables of this type within client modules. Similar exceptional rules apply to non-exported types that are used in the definition of exported types, and to non-exported types that are mentioned in the parameter lists of exported procedures.

Symbol files were introduced originally for accelerating the process of compilation. Consequently, their internal organization should aid the efficient transfer of the encoded interface data back into the symbol table of the compiler. In the past, this usually meant that the external representation of the interface on the symbol file more or less reflected the internal storage layout of the

corresponding symbol table, with additional provisions for mapping pointers within the table.

However, while a close conformity between the organization of storage in memory and on the symbol file was sensible in former times when processor speeds were not so great in comparison to storage access speeds, many of the assumptions about the relative cost of "calculation" versus that of "preservation" that underlie current symbol-file formats are no longer true. We are currently witnessing a rapidly widening gap between raw computing power on the one hand and speed of storage on the other hand at all levels of the storage hierarchy. As a consequence, it is increasingly becoming more efficient to recalculate a value that depends on some other values, rather than to store it on a file and read it back later. Experiments conducted by Griesemer [5] found an almost linear dependency between the length of a symbol file and the time required for reading it, regardless of the data encoding used on the file.

Moreover, most modern file systems offer some form of disk caching. If we compile whole series of modules regularly that share a common library base, and if all symbol files that are accessed more than once remain in the cache during such a compilation, this can affect performance dramatically. Since the size of a file cache is usually limited, choosing a more compact encoding for symbol files increases the number of files that can be held in the cache simultaneously.

With this in mind, we took a fresh look at the problem of finding an efficient symbol-file representation that is well suited for today's machines and will retain or even expand its benefits in the future. We might of course also have obtained smaller symbol files (and thereby higher file-access speeds) simply by incorporating into our compiler one of the established file-compression methods, such as the popular *LZW* technique [12]. However, we wanted to tackle the problem at its root rather than obscure it by the addition of yet another software layer to an already large compiler. Besides, the use of a general-purpose file-compression algorithm seemed to be somewhat inappropriate considering the fact that symbol files have a fairly regular structure. Moreover, most compression algorithms require a substantial amount of memory. It would have been unreasonable if, as an effect of compression, the symbol-file handler had turned out to dominate the memory requirements of the whole compiler, while it accounts for only a fraction of its size and run time.

We therefore set out to redesign the format of symbol files from scratch. This allowed us to specify an additional requirement for the new representation, namely *portability*, believing that this would make our work even more useful. The new symbol files were to contain no machine-dependent information, so that the

identical format could be used for all conceivable target architectures and the resulting symbol files would be interchangeable among different machines. Hopefully, this would simplify the enforcement of interface standards across a whole range of machines. Just as the provision of symbol files instead of textual interface definitions kept programmers from meddling with interface specifications, the distribution of (portable) symbol files by some central authority would encourage implementers of "standard modules" to develop in strict accord with the standard, instead of adding their own extensions for each specific target architecture. A portable symbol-file format would also allow us to reuse without modification the various tools that operate on symbol files, such as browsers.

We came up with four techniques aimed at reducing the size of the symbol files while simultaneously making them machine-independent: *compact and portable data representation*, *feature and token ordering*, *recalculation of dependent values*, and *recursive import propagation*. Each is discussed in detail below.

All four techniques have been incorporated into the compiler of a popular implementation of Oberon [2, 3], where they are now in everyday use. The resulting symbol files are highly compact and completely machine-independent, which is why we call them *universal*.

2.1 Compact and Portable Data Representation

Our symbol files encode constant data in a machine-independent and space-economical way. The two basic formats that are used for representing numbers and strings require a variable number of bytes on the file, and use a *stop bit* for denoting the last byte of a sequence. In order to keep matters simple, we purposefully represent each data value by an integral number of bytes, disregarding the potential further savings in size that might be possible if byte boundaries were transcended.

The scheme we employ for coding integers, suggested by Odersky [9], can be applied to values of any magnitude and is independent of the word length and the byte ordering of the machine used. Not only is the resulting file representation portable, but so is the algorithm, which can be used on any machine that offers 2's complement arithmetic. The following procedure in the programming language Oberon defines the representation used in our implementation, in which each encoded byte contains 7 data bits and a stop bit. Note that by Oberon's definition of the *DIV* and *MOD* operators, the expression $x \text{ DIV } 2^n$ for positive n is equivalent to an arithmetic right-shift of x by n positions, while the expression $x \text{ MOD } 2^n$ corresponds to clearing all but the n lowestmost bits of x . The opera-

Decimal	2's Complement Binary	Stop-Bit Encoding
-66	1...1011 1110	(1011 1110), (0111 1111)
-65	1...1011 1111	(1011 1111), (0111 1111)
-64	1...1100 0000	(0100 0000)
-1	1...1111 1111	(0111 1111)
0	0...0000 0000	(0000 0000)
1	0...0000 0001	(0000 0001)
63	0...0011 1111	(0011 1111)
64	0...0100 0000	(1100 0000), (0000 0000)
65	0...0100 0001	(1100 0001), (0000 0000)

Table 1. Examples of the variable-length number encoding.

tions occurring in the encoding algorithm therefore require no true multiplication or division, but can be performed by the simpler instructions "arithmetic shift" and "logical AND".

```

PROCEDURE WriteInt(x: INTEGER);
BEGIN
  WHILE (x < -64) OR (x > 63) DO
    Write(CHR(x MOD 128+128)); x := x DIV 128
  END;
  Write(CHR(x MOD 128))
END WriteInt;

```

The method requires only a single byte for representing the integers in the range [-64..63], two bytes for encoding the values in the intervals [-8192..-65] and [64..8191], three bytes for numbers in the ranges [-1048576..-8193] and [8192..1048575], and so on. Table 1 shows some examples of integers displayed as a decimal number, in a 2's complement binary representation, and as a succession of 8-bit bytes encoded by the algorithm above.

A similar stop-bit scheme can be also applied to identifiers and strings in order to suppress the termination character (with ordinal value zero) that appears at the end of every string in Oberon. Since the permissible range of ordinal values for characters in a string extends only from 1 to 127 in our compiler, we are free to use the high-order bits of characters to signal the end of a string. Many identifiers occurring in typical interface specifications are short, so that the space savings resulting from the use of a stop-bit representation can be considerable even if only a single byte is saved per string. For example, formal parameters and record fields often have single-letter names, so that removing an extra termination character halves the corresponding storage requirements.

The procedure below defines the encoding for strings employed in our implementation. Note that we use a high-order-bit value of 1 to signal the end of a string,

while a value of 0 is used for numbers. These particular choices for the position of the stop bit and its termination-signalling value simplify the algorithms used for reading.

```

PROCEDURE WriteString(s: ARRAY OF CHAR;
  len: INTEGER);
BEGIN
  IF len = 0 THEN Write(CHR(0))
  ELSE
    IF len > 1 THEN WriteBytes(s, len-1) END;
    Write(CHR(ORD(s[len-1])+128))
  END
END WriteString;

```

The constants of the remaining data types can be encoded space-efficiently by mapping them onto suitable integers. For example, a possible representation of the set $\{b_0, b_1, \dots, b_N\}$ is the integer $2^{b_0} + 2^{b_1} + \dots + 2^{b_N}$.

2.2 Feature and Token Ordering

Symbol files are not intended to be editable by human programmers. They are written and read solely by compilers and compiler-related tools. It is therefore quite acceptable to impose strict and complex syntax rules on the language in which symbol files are represented, rules that one could not reasonably expect any human to observe. Moreover, symbol files are usually read more often than they are written. Consequently, we may optimize their format for reading and neglect the expense of generating them.

Having studied several existing compilers, we observe that compiler writers seldom apply their expertise in the processing of artificial languages—manifest in the design of the compiler's parser—when they define a symbol-file format. Instead, they often use formats that are too simple and consequently uneconomical. By designing the format of a symbol file with the routine in mind that will later read it, one can easily find a representation that encodes an interface definition densely while nevertheless making it amenable to efficient parsing.

Our implementation uses a language for representing Oberon's interface definitions that can be parsed by a simple LL(1) mechanism. The key to the effectiveness of our scheme is *order*. The objects on our symbol file are ordered in a suitable way, as are the tokens (terminal symbols) of the language that describes them. By ordering the tokens of the language, the corresponding parser could be simplified considerably. The following discussion outlines our technique; a full specification of the symbol-file format is given in the Appendix.

In Oberon, as in most other languages of Algol ancestry, the symbol table contains entries for the *constants*, *variables*, *procedures*, and *types* appearing in declarations. Each constant, variable, and procedure has

a type associated with it, and types themselves may reference other types that have been used in their construction. When we linearize the symbol table, we therefore need to map multiple references to types in such a way that an isomorphic relationship can be established later when the corresponding symbol file is read back.

To this effect, our symbol-file reader maintains a table of types that have been input from the symbol file already. The symbol-file language contains a range of terminal symbols that represent the types in this table. On the symbol file, the structure of each type is specified exactly once, at which point a previously unused terminal symbol is associated with the new type. This old-type symbol is then used to denote all further occurrences of the type.

The features in the symbol table are sorted by their kind (constant, variable, ...) prior to being written to the file. This spares us the expense of identifying the kind of each feature individually. Furthermore, the terminal symbols of the symbol-file language are arranged in such a way that the start symbols of type references are easily distinguishable from all other terminal symbols in the language. This is achieved by reserving a continuous range of token values for them. Below is the framework of a grammar for a simple symbol-file language that can be parsed efficiently (*name* and *val* denote terminal strings and numbers).

```

SymFile = BEGIN
  { CONST { Type name val } }
  { VAR { [RDONLY] Type name } }
  { PROC { Typeres nameproc
    { [VAR] Typepar namepar } END } }
  { NEWTYP { Type } }
END.

Type = OldType | NewType

OldType = OLDTYP1
         | OLDTYP2
         | OLDTYP3
         | ...

NewType = NewArrayType
         | NewPointerType
         | NewRecordType
         | ...

```

Assuming that the start symbol for any production of *Type* is smaller than the token *CONST*, and that *CONST* < *VAR* < *RDONLY* < *PROC* < *NEWTYP* < *END*, then a simple parser for our symbol-file language can be structured in the manner shown below. The identifier *tag* denotes a global variable that contains the next symbol on the input stream (look-ahead of one symbol).

```

...
ReadInt(tag);

IF tag = CONST THEN ReadInt(tag);
  WHILE tag < VAR DO NEW(const);
    ReadTyp(const.typ); ReadString(const.name);
    (* read const.val depending on const.typ *);
    ReadInt(tag)
  END
END;

IF tag = VAR THEN ReadInt(tag);
  WHILE tag < PROC DO NEW(var);
    IF tag = RIDONLY THEN
      ReadInt(tag); var.readonly := TRUE
    ELSE var.readonly := FALSE
    END;
    ReadTyp(var.typ); ReadString(var.name);
    ReadInt(tag)
  END
END;
...

```

The symbol file is guaranteed to contain no syntax errors. A recursive-descent parser is therefore appropriate for parsing it, as this kind of parser can be implemented particularly elegantly and efficiently when syntax errors may be ruled out. The parsing of symbol files is a recursive process because the nonterminal symbol *Type* may appear on the right side of a production of *Type*; for example, an array consists of elements of another type. However, by choosing the ordering of the tokens for the *Type* productions carefully, the parser can be kept simple.

In the grammar above, type declarations appear at arbitrary points on the symbol file while the declaration order of all other features is defined strictly. This approach has been chosen purposefully. As an alternative, we might have written all types at the beginning of the symbol file, later followed by the constants, variables, and procedures that reference them. However, writing each first occurrence of a type inside of an object definition saves the space taken up by an "old-type" reference. Only those types that do not appear at least once in the declaration of another feature are specified at the end of the symbol file in a separate section.

Other seemingly arbitrary design decisions can contribute to the simplicity of the parser. For example, the objects describing parameters and record fields are usually linked together dynamically in the symbol table. We therefore store these on the symbol file in reverse order, making the construction of a linked list particularly easy during the reading process.

2.3 Recalculation of Dependent Values

The traditional approach to writing symbol files is to preserve all of the information that is available in the symbol-table handler about the exported features of a module. This usually includes a substantial amount of

derived data, i.e., knowledge about the features in the interface that is absent at the level of the source language, but computed by the compiler. Among this derived information are the addresses or entry numbers of variables and procedures, the relative offsets of record fields from the base address of a record, and the overall sizes of data types. Including this information in the symbol file spares the compiler from calculating it again when the symbol file is read again later.

However, recent developments in technology make this strategy questionable. Because of the rising ratio of processor speed versus storage-access time, it is becoming more economical to recalculate a dependent value than to store it on a file and read it back later. We have therefore elected not to include any compiler-derived information in our symbol files.

Our symbol files contain exactly the same information that is present in the source-level specification of an interface, while all derived information required by the symbol-table handler is reconstructed on the fly during reading. For example, the size of a record data type is calculated as the sum of the sizes of its constituents, plus some alignments. Since the dependent information may vary from target architecture to target architecture, equivalence of the source text and the symbol file is actually a prerequisite to machine independence. We also found that it results in higher overall compilation speed, regardless of the fact that more computing effort is required during reading.

We were able to isolate all machine-specific details of the recalculation step and remove them from the procedure that reads symbol files, making the symbol-table handler more easily retargetable. To this end, a procedure that belongs to the code generator is assigned to a procedure variable of the table handler. This procedure is up-called for each object that is read from the symbol file and subsequently computes all machine-dependent parameters of the object.

2.4 Recursive Import Propagation

Quite often an application is decomposed into several layers of modules that form an *import hierarchy*. The middle layers of this hierarchy serve a dual purpose: their modules act as libraries to the modules further up, while at the same time they are clients of the modules below them. When the interface declaration of such a middle-layer module is based on features that have themselves been imported from further down in the hierarchy, we speak of a *re-export* condition. Clients thereby become dependent on a re-exported module even when they do not import it directly.

In languages such as Modula-2 and Oberon, re-export conditions can occur only by way of *types* that have been imported from another module. For example, module *M1* in Figure 1 exports a variable *M1.v* that is

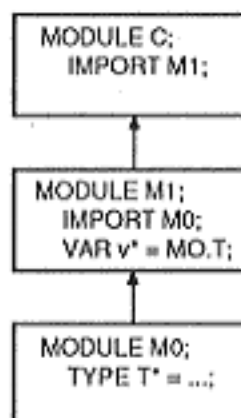


Figure 1. Module M1 re-exports M0 via a variable declaration.

declared as of type *M0.T*. *M0* therefore becomes visible in the interface of *M1*, so that changes in module *M0* may affect *M1*'s client module *C* regardless of the fact that *M0* is not mentioned in the import list of *C* explicitly.

There are two different approaches for the representation of re-export conditions in symbol files. As a seemingly straightforward solution, one may simply include in the symbol file the names of the modules that are re-exported. However, this makes the input of symbol files a recursive process, since the symbol-table handler requires access to the re-exported declarations before it can process the features that are based on them.

In the past, most implementers have opted for another approach instead and have used *self-contained symbol files* [4, 5, 6, 16]. A self-contained symbol file includes the complete definition of every re-exported feature, even of features re-exported through several layers of the module hierarchy, thereby flattening this hierarchy from the viewpoint of client modules. In the example above, the symbol file of module *M1* would contain the full description of *M0.T*, but not of any other object that might also be exported by *M0*.

According to Wirth [15], one of the major reasons for the use of self-contained symbol files in the past was the limited size of main storage available to the symbol-table handler. Using self-contained symbol files, one imports only those features indirectly that are actually needed during a compilation, while otherwise one imports all the features exported by each indirectly imported module. However, the memory requirements of the symbol-table handler can nowadays be fulfilled even by desktop computers and should no longer play a role in the design of the symbol-file format.

The other advantage of self-contained symbol files concerns the number of files that need to be read during a compilation. Self-contained symbol files flatten the module hierarchy, so that the compiler can avoid

processing the symbol files of modules that are imported indirectly. In our experience, however, few modules are imported indirectly without also being imported directly. On the other hand, making symbol files self-contained not only adds to their structural complexity, but also to their size. For reasons already mentioned, we find that we prefer compactness over self-sufficiency, while we can almost disregard the total number of files that are accessed, due to the effects of file caching. We have therefore reverted to the concept of recursive imports and abandon the principle that all symbol files are self-contained.

Although symbol files are imported recursively in our compiler, this does not imply that the symbol-table handler needs to be fully re-entrant, which would make it less efficient because all symbol-table operations would have to be parametrized. For example, imported features are normally added to a module's *global scope*, which can be anchored in a global variable only if module scopes are read one at a time. However, by separating the information about re-exported modules from the actual symbol data, reading a module's symbol file can proceed without interruption.

In our implementation, the names of all re-exported modules are listed at the beginning of a symbol file, allowing the recursive import of re-exported modules before the scope of the current module is even opened. As a disadvantage of this strategy, the re-export information needs to be separated from the scope data. However, this can be done in a single traversal of the symbol table simply by employing two separate intermediate buffers, which are later combined to form the symbol file.

3. A Practical Example

The Oberon program text below contains all of the declarations necessary for a module that manages files on a random-access storage. A similar module, along with a detailed description of its implementation can be found in [16]. Note that the pointer type *File* is exported, but the record it references is not. The components of the *Rider* data structure are only partially exported.

```

MODULE MFiles;

TYPE
  FileName* = ARRAY 32 OF CHAR;
  File* = POINTER TO Header;

  Rider* = RECORD
    eof*: BOOLEAN;
    res*: LONGINT;
    file: File;
    pos, adr: LONGINT
  END;
  
```

```

Header = RECORD (*1k allocation blocks*)
  name: FileName;
  len: LONGINT;
  soc: ARRAY 245 OF LONGINT
END;

PROCEDURE Old*(name: FileName): File;
PROCEDURE New*(name: FileName): File;
PROCEDURE Register*(f: File);
PROCEDURE Length*(f: File): LONGINT;
PROCEDURE Set*(VAR r: Rider; f: File;
               pos: LONGINT);
PROCEDURE Read*(VAR r: Rider; VAR x: CHAR);
PROCEDURE Write*(VAR r: Rider; x: CHAR);

END MFiles.

```

While it is obvious that universal symbol files have to contain a detailed description of all components of a record, including the nonexported ones, one might assume that traditional symbol files could omit this

information. Unfortunately, this is not the case in systems that offer garbage collection. Garbage collection requires that all pointers can be located and traversed. The record type *Rider* in our example contains an invisible pointer to a file. This knowledge needs to be present on the symbol file, as one might declare an extension of the type *Rider* in another module, whose *file* field must be traversed also. Consequently, a note must be made even in a traditional symbol file that a hidden pointer is present in the *Rider* type, and its offset has to be specified.

The complete contents of the universal symbol file for module *MFiles* are displayed Figure 2 along with some comments. Note that invisible record fields have the empty string as their name rather than an explicit "invisible" flag. In the comment section on the right, we have marked in boldface each point at which a previously unused "old-type" symbol is associated with a type definition that has just been completed.

Symbol File Representation

```

BEGIN 9C2E7EA9H " "
PROC
  NOTYP "Write"
  CHAR "x"
  VAR RECORD
    NOTYP "Rider"
    LONGINT "*"
    LONGINT "*"
    POINTER
    INVISIBLE RECORD
      NOTYP "Header"
      ARRAY
        LONGINT "*" 245
        "*"
      LONGINT "*"
      ARRAY
        CHAR "FileName" 32
        "*"
      END
      "File"
    "*"
    LONGINT "res"
    BOOLEAN "eof"
  END
  "r"
END
NOTYP "Set" LONGINT "pos" OLDTYP2 "f" VAR OLDTYP1 "r" END
NOTYP "Register" OLDTYP2 "f" END
NOTYP "Read" VAR CHAR "x" VAR OLDTYP1 "r" END
OLDTYP2 "Old" DYNARR CHAR "*" "name" END
OLDTYP2 "New" DYNARR CHAR "*" "name" END
LONGINT "Length" OLDTYP2 "f" END
END

```

Comments

```

version-stamp, no imports
no constants or variables
Proc Write() returning no value
Write Par2 "x"
Write Par1 "r"
Type Rider =: OLDTYP1
Rider Field5 invisible
Rider Field4 invisible
Rider Field3 invisible
Type File =: OLDTYP2
Type Header =: OLDTYP3
Header Field3 invisible
Unnamed Array Type =: OLDTYP4
Header Field3 invisible
Header Field2 invisible
Header Field1 invisible
Type FileName =: OLDTYP5
Header Field1 invisible
Type Header
Type File
Rider Field3 invisible
Rider Field2 "res"
Rider Field1 "eof"
Type Rider
Write Par1 "r"
Proc Write
Proc Set() returning no value
Proc Register() returning no value
Proc Read() returning no value
Proc Old(): TYP2, ... =: OLDTYP6
Proc New(): TYP2, ... =: OLDTYP7
Proc Length(): LONGINT

```

Figure 2. The universal symbol file for module *MFiles*.

4. Results

We have combined the four techniques described above, *compact and portable data representation, feature and token ordering, recalculation of dependent values, and recursive import propagation*, in an Oberon compiler for the MacOberon environment [2, 3]. This compiler has been widely disseminated and has demonstrated its usefulness and robustness convincingly. In order to evaluate our approach, we will compare it in the following to a variation of Gutknecht's method of writing symbol files [6], which was used in a previous version of MacOberon's compiler. Both symbol-file formats are specified in the Appendix.

Gutknecht's symbol-file format is suited especially well for a comparison, as it has been published in detail and a great number of existing compilers for various target machines employ it with only slight modifications. While the algorithms for reading and writing symbol files in this format are portable, the resulting files are not. They contain machine-specific data such as *addresses, offsets, and sizes*, and the encoding of this information furthermore depends on the byte ordering of the target machine.

The following benchmarks were carried out under MacOberon Version 3.10 on a Macintosh II computer (MC68020 processor running at 16MHz), which is a relatively slow computer and therefore makes the job of measuring performance easy. As we have mentioned above, the advantages of our scheme become even more apparent with rising processor speed, although the effect was less pronounced than we had expected when we ran some tests on faster Macintosh models, most probably because these newer computers also have faster disk controllers. However, the margin of measuring error is greater on a faster processor because the unit of machine time against which our measurements are taken is the same on all models.

Three different sets of modules were used in the benchmarks. The first group of seven modules in the tests below form the "outer core" of the Oberon operating system, implementing its user interface. The second and third groups of modules represent typical application packages, namely the Write document processing system [10], and the Draw object-oriented graphics editor [16].

Table 2 gives an impression of the compactness of our representation. It compares the sizes (in bytes) of universal symbol files ("New") with the sizes of symbol files obtained when the identical interface is compiled by a previous version of the MacOberon compiler that uses Gutknecht-style symbol files ("Old").

Our remaining two tables report some timing results. The first two columns of Table 3 present the times (in milliseconds) that the compiler expends on rebuilding the information contained in the interfaces of imported

Module	Old SF Size	New SF Size	New : Old
Texts	2196	1073	0.49
Viewers	653	285	0.44
Oberon	2157	955	0.44
MenuViewers	450	149	0.33
TextFrames	2126	955	0.45
System	333	250	0.75
Edit	143	107	0.75
WriteFrames	3207	1364	0.43
WritePrinter	878	325	0.37
ParcElems	2085	352	0.17
Write	261	197	0.75
WriteTools	1256	613	0.49
Graphics	2490	1147	0.46
GraphicFrames	1590	383	0.24
Draw	182	140	0.77
Rectangles	710	81	0.11
Curves	728	98	0.13
Splines	743	93	0.13
Σ	22188	8567	0.39

Table 2. Symbol file sizes in bytes.

modules. In the old compiler (Gutknecht-style symbol files), this is equal to the total time spent within the procedure that reads symbol files, while in the new compiler (universal symbol files) the time needed to reconstruct derived information has been added also. Columns three and four indicate how many symbol files are opened during compilation of each module. Since universal symbol files are not self-contained, more files need to be opened in general in order to read features that are imported indirectly.

Table 4 puts the symbol-file reading times in relation to the total time required for the compilation of each module. While its first two columns replicate information also found in the previous table, its third column lists the time (in milliseconds) required for the remaining steps of compilation, i.e., without reading the symbol file. The total compilation time for each module can be calculated by adding to this figure the corresponding value from column one or two. The fourth and fifth columns represent the quotient of the import time to the total compilation time.

It is notable that the proportion of compilation time expended on symbol-file reading varies greatly. For some small modules that mainly implement extensions of other modules, e.g., *Rectangles*, a third of the total compilation time is spent reading symbol files.

In the benchmarks above, files were closed and the corresponding file buffers flushed after the compilation of each module. Compiling the modules in succession without flushing file caches tilts the performance balance significantly in favor of our scheme because it

Module	Old Imp. Time	New Imp. Time	Old #Mod Imp.	New #Mod Imp.
Texts	210	210	5	6
Viewers	60	30	1	1
Oberon	550	400	8	9
MenuViewers	200	260	4	7
TextFrames	580	480	9	11
System	730	510	11	12
Edit	550	350	9	9
WriteFrames	710	510	10	12
WritePrinter	600	460	7	9
ParcElems	550	410	8	10
Write	860	510	13	14
WriteTools	710	430	9	10
Graphics	430	310	7	9
GraphicFrames	480	430	8	11
Draw	660	510	10	13
Rectangles	550	450	8	12
Curves	450	410	6	11
Splines	650	530	7	12
Σ	9530	7200	140	178

Table 3. Import times in milliseconds and number of imported modules processed.

reduces the latency of file access without reducing individual read operations much. Moreover, because our symbol files are much shorter, more of them are likely to be kept in the file cache.

5. Universal Version Keys

At the time of linking, all import/export relationships between modules need to be resolved. For this purpose, each client module should in principle contain a list of

the features it imports from various libraries, while each corresponding library module should contain a list of the features it exports. However, apart from the space that would be required in each client module for describing the imported features in detail, this would also necessitate a fine-grained comparison to ensure that all of the features required by a client are actually present in the version of the library encountered during linking. Such a comparison is time-consuming and therefore not well suited for systems offering dynamic loading.

Module	Old Imp. Time	New Imp. Time	Rest Comp. Time	Old I:(R+I)	New I:(R+I)
Texts	210	210	4120	0.05	0.05
Viewers	60	30	1350	0.04	0.02
Oberon	550	400	2250	0.20	0.15
MenuViewers	200	260	1440	0.12	0.15
TextFrames	580	480	5000	0.10	0.09
System	730	510	3170	0.19	0.14
Edit	550	350	2500	0.18	0.12
WriteFrames	710	510	8100	0.08	0.06
WritePrinter	600	460	2690	0.17	0.14
ParcElems	550	410	3700	0.13	0.10
Write	860	510	3570	0.19	0.13
WriteTools	710	430	4980	0.12	0.08
Graphics	430	310	3390	0.11	0.08
GraphicFrames	480	430	2980	0.14	0.13
Draw	660	510	1540	0.30	0.25
Rectangles	550	450	1080	0.34	0.29
Curves	450	410	1690	0.21	0.20
Splines	650	530	1330	0.33	0.28
Σ	9530	7200	55080	0.15	0.12

Table 4. Import times in milliseconds compared to total compilation time.

However, a similar comparison already takes place during separate compilation, a fact that can be exploited by *approximate matching* of clients to libraries with the help of *version keys*. During the separate compilation of a library module, the compiler assigns such a version key to the library's interface and includes it in the symbol file. When client modules are separately compiled later, the compiler verifies that the client is consistent with the library and includes the key of the library in the object file of the client. At the time of linking, the key on the client's object file must match the key of the library; otherwise the client has been compiled against a different version of the library and they cannot be linked. The complex checks of compatibility between libraries and clients that were performed at the time of compilation therefore need not be repeated during linking.

Version-key schemes are very popular in systems that support separate compilation, and many existing implementations [4, 6, 16] use the *time stamp* of compilation to uniquely identify a specific version of an interface. However, time stamps are practical as version keys only in a universal module space in which each module is associated with a site of origin. Time stamping makes it impossible to carry out an agreed change in a certain library module on several different computers independently while maintaining interchangeability of client modules. Instead, one has to generate the new symbol file on a single machine and distribute it to all other sites so that the time stamps are identical everywhere.

On the other hand, the use of *fingerprints* instead of time stamps is able to yield symbol files that are not only universal with respect to representation of their content, but also with respect to the identification of specific versions. Consequently, these files are completely interchangeable between different machines. A fingerprint is a scalar value that is computed deterministically from the contents of an interface by a hashing function. In contrast, time stamps do not depend on the contents of interfaces at all. Fingerprinting is more flexible, since identical interfaces yield identical fingerprints and thereby identical symbol files regardless of the compilation environment. This enables close control over compatibility in the implementation of software systems that are to be portable to several different target architectures. It also makes interface changes easily reversible, a property that we have found to be much appreciated by programmers.

Fingerprints also simplify the detection of changes in an interface. Such changes are often unwanted because they invalidate all clients of the module affected. Consequently a user should have the option to prevent inadvertent interface changes. This can be achieved by

treating such changes as compilation errors unless a certain compiler option is enabled.

Gutknecht [7] mentions that the task of detecting equivalence between an existing interface and a newly created one is a challenge and suggests two different strategies for this purpose. While the more complex of these strategies involves a full structural comparison between two symbol tables, the simpler alternative requires that symbol files always be created in some canonical form. Equivalence of an old and a new interface can then be tested by comparing the corresponding symbol files byte by byte. Fingerprinting requires only half as much effort as Gutknecht's second solution, because only the new interface has to be read completely, yielding a fingerprint that can then be compared to the fingerprint of the previous version of the interface. There is no need to process the existing symbol file a second time, which is costly as it involves file operations.

Time stamps are probably slightly more secure than fingerprints of equivalent length because the chances of inadvertently compiling a module concurrently on different machines are very small, while the safety of fingerprints depends on the quality of the fingerprinting function. However, this disadvantage can be countered simply by increasing the number of bits used in the representation of the fingerprint. In the extreme case, the complete interface in its canonical form may serve as the fingerprint, yielding absolute safety.

5.1 Simple Fingerprints

A fingerprinting function computes a scalar value from a structured interface. We require it to have the property that the change of a single bit in the interface must change the value of the fingerprint. Note that although an interface is really an unordered *set* of features, the ordering is significant in the linear representation used on symbol files. This is because clients reference imported features by their *relative position* on the symbol file of a library. Consequently, changing the ordering of features on a symbol file must lead to a change in the corresponding fingerprint. However, this does not imply a restriction for the programmer, because the compiler can create symbol files in a canonical form in which the features are sorted, so that the ordering at the level of the programming language need not be reflected on the symbol file.

The only important requirement for a fingerprinting algorithm in practice is that interfaces that differ only slightly be guaranteed to obtain different fingerprints. It doesn't really need to concern us if completely different interfaces by chance obtain the same fingerprint. A simple fingerprinting function that in our experience yields good results, while it can be implemented efficiently, is the following:

```

PROCEDURE Fingerprint(interface: STREAM OF BYTE;
  VAR fp: ARRAY N OF BYTE);
  VAR i, j: INTEGER; ch: BYTE;
BEGIN i := 0; j := 0; fp := [0, 0, ..., 0];
  WHILE ~ (last byte of interface processed) DO
    ch := (next byte of interface);
    fp[j] := (fp[j] + ROT(ch, i)) MOD 256;
    INC(j); IF j=N THEN INC(i); i:=0 END
  END
END Fingerprint;

```

The function *ROT(x: BYTE; n: INTEGER)* in the algorithm above performs a bitwise left-rotation of its first argument by the number of bit positions given in the second argument. It is introduced to identify transpositions of a multiple of *N* byte positions in the interface, which would otherwise be undetectable.

The algorithm given above is our own ad-hoc solution to the fingerprinting problem; we have not subjected it to theoretical analysis. For example, the probability of collisions for well-formed interfaces is unknown. However, a compiler incorporating this algorithm has been in widespread use for some time without any reports of fingerprint collisions. With some confidence, we therefore conclude that an algorithm as simple as this one is reasonably well suited for fingerprinting purposes in practice. Its safety can be varied as needed by adjusting the value of the constant *N*. We find that a fingerprint of four bytes in length is adequate, providing sufficient security while ensuring easy management.

6. Summary and Conclusion

The ongoing improvements in computing hardware have reached a point at which a complete reassessment of storage schemes is in order. We have substituted a more up-to-date symbol-file format for a proven but outdated format whose design at the time had been motivated largely by hardware constraints such as scarcity of memory. While the new format is more complex than its predecessor, it is based on the theory of context-free languages and can be parsed efficiently using recursive-descent methods.

The immediate result of our technique is reduced storage requirement for the symbol file, along with a worst-case performance that is comparable to the previous method. In typical cases, for example, when disk caching is used, much better performance of our scheme can be expected. Moreover, technology is evolving in favor of our solution. As time goes by and CPU speeds increase further in relation to disk-access times and transfer rates, our format will become even more attractive.

Equally important, universal symbol files contain no target-machine-specific information. Although pro-

visions have been made in the grammar to allow for the inclusion of such extensions as in-line code procedures for a specific architecture, the basic format of symbol files that describes a plain Oberon interface (i.e., one that does not import the features of module *SYSTEM*) is identical for all machine types. This is a prerequisite to symbol-file portability, which is likely to bring about improvements of organizational nature in heterogeneous environments.

Acknowledgements: The author would like to thank Niklaus Wirth for teaching him how to build compilers, and for his valuable criticisms regarding this paper. He also gratefully acknowledges the contribution of the anonymous referees, whose comments helped to improve the presentation of this material.

References

1. Foster DG (1986) Separate Compilation in a Modula-2 Compiler. *Software—Practice and Experience*, 16, 2, 101–106
2. Franz M (1990) The Implementation of MacOberon. Departement Informatik, ETH Zürich, Report No. 141
3. Franz M (1993) Emulating an Operating System on Top of Another. *Software—Practice and Experience*, 23, 6, 677–692
4. Geissmann L (1983) Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith. ETH Zürich, Dissertation No. 7286
5. Giesecke R (1991) On the Linearization of Graphs and Writing Symbol Files. Departement Informatik, ETH Zürich, Report No. 156
6. Gutknecht J (1986) Separate Compilation in Modula 2: An Approach to Efficient Symbol Files. *IEEE Software*, 3, 6, 29–38
7. Gutknecht J (1989) Variations on the Role of Module Interfaces. *Structured Programming*, 10, 1, 40–46
8. Mitchell JG, Maybury W, Sweet R (1979) Mesa Language Manual, Version 5.0, CSL-79-3. Xerox Corporation, Palo Alto Research Center, Systems Development Department
9. Odersky M (1989) Private Communication.
10. Szyperki CA (1992) Writing Applications: Designing an Extensible Text Editor as an Application Framework. *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92)*, Dortmund, 247–261
11. United States Department of Defense (1980) Reference Manual for the Ada Programming Language.
12. Welch TA (1984) A Technique for High-Performance Data Compression. *IEEE Computer*, 17, 6, 8–19
13. Wirth N (1982) *Programming in Modula-2*. Springer-Verlag, Berlin
14. Wirth N (1988) The Programming Language Oberon. *Software—Practice and Experience*, 18, 7, 671–690
15. Wirth N (1993) Private Communication.
16. Wirth N, Gutknecht J (1993) *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, Wokingham

Appendix A. Format of Original Oberon Symbol Files

SymFile	=	BEGIN MOD key name {Element}.	
Element	=	MOD key name	<i>imported module</i>
		CONST type value name	<i>constant</i>
		(TYPE HDTYPE) type mod name	<i>type, hidden type</i>
		(VAR RVAR) type varno name	<i>variable, read-only variable</i>
		(FLD RFLD) type offset name	<i>field, read-only field</i>
		(VALPAR VARPAR) type adr name	<i>parameter, VAR parameter</i>
		PLIST (Element) _{par} PRO type _{res} procno name	<i>exported procedure</i>
		POINTER type _{record} mod	<i>pointer type</i>
		PLIST (Element) _{par} PROCTYP type _{res} mod	<i>procedure type</i>
		ARRAY type _{elem} mod size	<i>array type</i>
		DYNARR type _{elem} mod size lenOff	<i>dynamic array type</i>
		FLIST (Element) _{id} RECORD type _{base} mod size descno	<i>record type</i>
		HDPTR offset	<i>hidden pointer field</i>
		FIXUP type _{ptr} type _{record}	<i>fix pointer's declaration</i>
		SYSTEM type sysflag.	<i>mark type as special</i>

Names are sequences of characters terminated by 0X. All other lower case identifiers denote numbers.

Type definitions always precede their use, except for pointer declarations that may be part of recursive data structures. In this case, the pointer is initially defined to point to an illegal base type, and a fixup is made after the definition of the pointer's base type.

There are predefined type numbers for the basic types *BOOLEAN*, *CHAR*, *SHORTINT*, *INTEGER*, *LONGINT*, *REAL*, *LONGREAL*, *SET*, *STRING*, *NOTYP*, and *NILTYP*, the latter two denoting the result type of a proper procedure (i.e., one that returns no function result) and the type of the constant *NIL*.

Appendix B. Format of Universal Oberon Symbol Files

SymFile	=	BEGIN key {name _{imp} key _{imp} } ** [CONST { Type name val }] [VAR { (RDONLY) Type name }] [PROC { Type _{res} name [(VAR) Type _{par} name] END }] [ALIAS { Type name }] [NEWTYP { Type }] END.	<i>key, imported modules</i> <i>constants</i> <i>variables</i> <i>exported procedures</i> <i>type-aliases</i> <i>types not previously output</i>
Type	=	BasicType [Module] OldType [SYSTEM] [HIDDEN] NewType	<i>basic type</i> <i>previously defined</i> <i>first occurrence</i>
Module	=	MOD1 MOD2 ... MOD32 MOD modno	<i>modno implicit</i> <i>modno explicit</i>
BasicType	=	BOOL CHAR SHORTINT INTEGER ...	<i>basic types of Oberon</i>
OldType	=	OLDTYP1 OLDTYP2 OLDTYP3 ... OLDTYP99 ...	<i>defined implicitly in NewType</i>
NewType	=	DYNARR Type _{elem} name ARRAY Type _{elem} name interval POINTER Type _{base} name RECORD Type _{base} name [(RDONLY) Type _{id} name] END PROCTYP Type _{res} name [(VAR) Type _{par} name] END	<i>dynamic array type</i> <i>array type</i> <i>pointer type</i> <i>record type</i> <i>procedure type</i>

The terminal symbols are ordered as follows, which simplifies the parsing process:

```
... OLDTYP99 < ... < OLDTYP3 < OLDTYP2 < OLDTYP1
< BOOL < CHAR < SHORTINT < INTEGER < LONGINT
< REAL < LONGREAL < SET < STRING < NOTYP < NILTYP
< MOD1 < MOD2 < MOD3 < ... < MOD32 < MOD
< DYNARR < ARRAY < POINTER < RECORD < PROCTYP
< SYSTEM < HIDDEN < RDONLY
< CONST < VAR < PROC < ALIAS < NEWTYP
< END
```