

Technological Steps toward a Software Component Industry

Michael Franz

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

Abstract. A *machine-independent abstract program representation* is presented that is twice as compact as machine code for a CISC processor. It forms the basis of an implementation, in which the process of code generation is deferred until the time of loading. Separate compilation of program modules with type-safe interfaces, and dynamic loading (with code generation) on a per-module basis are both supported.

To users of the implemented system, working with modules in the abstract representation is as convenient as working with native object-files, although it leads to several new capabilities. The combination of portability with practicality denotes a step toward a *software component industry*.

1. Introduction

The rapid evolution of hardware technology is constantly influencing software development, for better as well as for worse. On the downside, faster hardware can conceal the complexity and cost of badly-designed programs; Reiser [Rei89] is not far off the mark in observing that sometimes "software gets slower more quickly than hardware gets faster". On the other hand, improvements in hardware, such as larger memories and faster processors, have also provided the means for better software tools:

Often enough, methodical breakthroughs have been indirect consequences of better hardware. For instance, software-engineering techniques such as *information hiding* and *abstract data types* could only be developed after computers became powerful enough to support compilers for modular programming languages. Mechanisms such as *separate compilation* place certain demands on the underlying hardware and so had a chance of proliferation only after sufficiently capable computers were commonplace.

This paper presents another example of a systematic technological advance that owes its viability to improved hardware. The first part of the paper describes a technique for representing programs abstractly in a format that is twice as compact as object code for a CISC processor. Combined with the speed of current processors and abundance of main storage, the use of this intermediate representation makes it possible to accelerate the process of code generation to such a degree that it can be performed on-the-fly during program loading on ordinary desktop computers, even with a high resulting code quality.

A system has been implemented that permits the convenient use of program modules in this machine-independent intermediate representation, as if they had been compiled to native code. The second part of the paper discusses some of the possibilities that arise from this scenario, reviews related work, and concludes in an outlook to future developments.

2. Abstract Program Representation

2.1. Intermediate Languages

Ever since the late 1950's, there have been attempts to design a *universal computer-oriented language* (UNCOL) that would be powerful enough so that programs originating in any problem-oriented language could be translated into it, and code for any processor architecture could be generated from it [SWT58]. Having such a language would enable us to construct compilers for m languages to be run on n machines by writing only $m+n$ programs (m front-ends and n code generators), instead of $m \times n$.

To this date, no proposed UNCOL has been met with general agreement. However, many compilers have employed some form of *intermediate language* (IL) on the path from source code to final executable code image. Such an IL is distinguished from the transient data structures found in memory between different passes of a compiler by providing a stand-alone program representation that can be stored on a data file.

A straightforward method for obtaining an IL representation of a program is to express it as an instruction sequence for some *fictitious computer*, also called an *abstract machine* [PW69]. Most ILs, including the very first UNCOL ever proposed [Con58], follow this pattern, and many implementations, such as *BCPL* [Ric71], *SNOBOL4* [Gri72], and *Pascal-P* [NAJ76], have been based on abstract machines. Besides abstract machines, ILs have also been based on linearized parse-trees [GF84, DRA93a]. Furthermore, there are compilers that compile via ordinary high-level programming languages by way of source-to-source program translation [ADH89].

This paper introduces an IL approach that results in a highly compact intermediate representation. I call it *semantic-dictionary encoding* (SDE). SDE preserves the full semantic context of source programs while *adding further information* that can be used for accelerating the speed of code generation. SDE forms the basis of an implementation, in which a code generator has been combined with the module loader, forming a *code-generating loader*.

2.2. An Overview of Semantic-Dictionary Encoding

SDE is a dense representation. It encodes a syntactically correct source program by a succession of indices into a *semantic dictionary* (SD), which in turn contains the information necessary for generating native code. The dictionary itself is not part of the SDE representation, but is constructed dynamically during the translation of a source program to SDE form, and reconstructed before (or during) code generation. This method bears some resemblance to commonly used data compression schemes [Wei84].

With the exception of wholly constant expressions, SDE preserves all of the information that is available at the level of the source language. Hence, unlike abstract-machine representations, transformation to SDE preserves the *block structure* of programs, as well as the *type* of every expression. Moreover, when used as the input for code-generation, SDE in certain cases provides for *short-cuts* that can increase translation efficiency.

A program in SDE form consists of a *symbol table* (in a compact format, as explained in [Fra93b]) and a series of *dictionary indices*. The symbol table describes the names and the internal structure of various entities that are referenced within the program, such as *variables*, *procedures*, and *data types*. It is used in an initialization phase, in the course of which several *initial entries* are placed into the SD. The encoding of a program's actions

consists of references to these initial dictionary entries, as well as to other entries added later in the encoding process.

Dictionary entries represent nodes in a directed acyclic graph that describes the semantic actions of a program abstractly. In its most elementary form, an SD is simply such an *abstract syntax-tree* in tabular shape, in which the references between nodes have been replaced by table indices. Each dictionary entry consists of a *class* attribute denoting the semantic action that the entry stands for (e.g., *assignment*, *addition*, etc.), and possibly some references to objects in the symbol table and to other dictionary entries.

What differentiates a tabular abstract syntax-tree from an SD is that the latter can describe also *generic characteristics* of *potential nodes* that might appear in such a tree. In addition to *complete entries* that directly correspond to nodes of an abstract syntax tree, semantic dictionaries contain also *generic*, or *template entries*. These templates have the same structure as complete entries, but at least one of their attributes is missing, as recorded in a *status* flag. In SDE, complex program statements can be represented not only by complete entries, but also by templates in combination with other entries. For example, the statement $a + b$ can be represented by the index of an "addition" template followed by the indices of two variable-reference entries.

Now suppose that a template existed in the SD for every construct of the source language (*assignment*, *addition*, etc.), and that furthermore the SD were initialized in such a way that it contained at least one entry (possibly a template) for every potential use of every object in the symbol table. For example, an object describing a *procedure* in the programming language *Oberon* [Wir88] requires a minimum of four entries in the dictionary, relating in turn to a *call* of the procedure, *entry* of the procedure, *return* from the procedure, and *addressing* the procedure, which is used in the assignment of the procedure to procedure variables. There are no other operations involving procedures in Oberon.

These preconditions can be fulfilled by initializing the SD in a suitable way. They enable us to represent any program by only a symbol table and a succession of dictionary indices. As an example, consider the following module *M* in the programming language Oberon:

```
MODULE M;

  VAR i, j, k: INTEGER;

  PROCEDURE P(x: INTEGER): INTEGER;
  BEGIN ...
  END P;

BEGIN
  i:=P(i); i:=i+j; j:=i+k; k:=i+j; i:=i+j
END M.
```

In order to encode this program by the method of SDE, we first need to initialize the SD. Initialization depends on the source language (Oberon in this case) and on the objects in the symbol table. The symbol table for the example program contains three integer variables (i , j , and k) and a function procedure (P). After initialization, the corresponding SD might look like the following (the individual entries' indices are represented by symbolic names so that they can be referenced further along in this paper, and missing attributes of templates are denoted by a dot).

Index	Class	Meaning	Status
asgn	assignment	. := .	both arguments missing
plus	addition	. + .	both arguments missing
...
vi	variable	i	complete
vj	variable	j	complete
vk	variable	k	complete
refp	address	P	complete
callp	function call	P(.)	argument missing
entp	entry	P-BEGIN	complete
retp	return	P-RETURN .	argument missing
...

The instruction sequence that constitutes the body of module M may then be represented by the following sequence of 24 dictionary indices:

asgn	vi	callp	vi	$i := P(i)$
asgn	vi	plus	vi vj	$i := i + j$
asgn	vj	plus	vi vk	$j := i + k$
asgn	vk	plus	vi vj	$k := i + j$
asgn	vi	plus	vi vj	$i := i + j$

This is where the second major idea of SDE comes in. What if we were to keep on *adding* entries to the dictionary during the encoding process, based on the expressions being encoded, in the hope that a *similar expression* would occur again later in the encoding process? For example, once we have encoded the assignment $i := P(i)$ we might add the following three entries to the dictionary:

Class	Meaning	Status
function call	P(i)	complete
assignment	i := .	right-side argument missing
assignment	i := P(i)	complete

Thereafter, if the same assignment $i := P(i)$ occurs again in the source text, we can represent it by a single dictionary index. If another assignment of a different expression to the variable i is come across, this may be represented using the template "assign to i ", resulting in a shorter encoding. In this manner, the body of module M can be encoded by only 16 dictionary indices, instead of the previous 24. This shorter encoding is shown below, along with the entries added to the SD in the process (assuming that, after initialization, the dictionary comprised $n-1$ entries).

asgn	vi	callp	vi	$i := P(i)$
n+1		plus	vi vj	$i := i + j$
asgn	vj	n+3	vk	$j := i + k$
asgn	vk	n+5		$k := i + j$
n+6				$i := i + j$

Index	Class	Meaning	Status
...
n	function call	P(i)	complete
n+1	assignment	i := .	right-side argument missing
n+2	assignment	i := P(i)	complete
n+3	addition	i + .	right-side argument missing
n+4	addition	. + j	left-side argument missing
n+5	addition	i + j	complete
n+6	assignment	i := i + j	complete
n+7	addition	. + k	left-side argument missing
n+8	addition	i + k	complete
n+9	assignment	j := .	right-side argument missing
n+10	assignment	j := i + k	complete
n+11	assignment	k := .	right-side argument missing
n+12	assignment	k := i + j	complete
...

2.3. Increasing the Speed of Code Generation

The decoding of a program in SDE form is similar to the encoding operation. At first, the dictionary is initialized to a state identical to that at the onset of encoding. Since the symbol table is part of the SDE file-representation, the decoder has all required information available to perform this task. Thereafter, the decoder repeatedly reads dictionary indices from the file, looking up each corresponding entry in the SD. Whenever a complete entry is found in this manner, its meaning is encoded directly in the dictionary and the decoder can proceed to process the next index on the input stream. If a template is retrieved instead, it is copied, further entries corresponding to its undefined attributes are input in turn, and the modified copy is added to the dictionary as a new complete entry. Moreover, additional templates are sometimes added to the SD according to some fixed heuristics, in the hope that a corresponding branch of the program's syntax tree will show up further along.

The interesting fact now is that SDE actually provides *more explicit information* than the source text, namely about *multiple textual occurrences of identical subexpressions* (including, incidentally, *common designators*). This can sometimes be exploited during code generation, resulting in an increase in the speed of code output. Consider again the (second, more compact) SDE representation of module *M* above. Now suppose that we want to generate object code for a simple stack machine directly from this SDE form. Let us assume that we have processed the first two statements of *M*'s body already, yielding the following instruction sequence starting at address *a*:

a	LOAD	i	<i>load i onto stack</i>
a+1	BRANCH	P	<i>call procedure P with argument i</i>
a+2	STORE	i	<i>assign result to i</i>
a+3	LOAD	i	<i>load i</i>
a+4	LOAD	j	<i>load j</i>
a+5	ADD		<i>add i to j</i>
a+6	STORE	i	<i>assign their sum to i</i>

Let us further assume that we have kept a note in the decoder's SD, describing which of the generated instructions correspond to what dictionary entry. For example, we might simply have recorded the program counter value twice for every entry in the SD, both before and after generating code for the entry:

Index	Class	Meaning	Begin	End	Invar
...
n	function call	$P(i)$	a	a+1	No
n+1	assignment	$i := P(i)$	a	a+2	No
...
n+5	addition	$i + j$	a+3	a+5	Yes
n+6	assignment	$i := i + j$	a+3	a+6	Yes
...

A setup such as this allows us to bypass the usual code generation process under certain circumstances, replacing it with a simple *copy operation* of instructions already generated. For example, when we encounter the second reference to entry $n+6$ in the semantic-dictionary encoding of module M , we know that we have already compiled the corresponding statement $i := i + j$ earlier on. In this case, we may simply re-issue the sequence of instructions generated at that earlier time, which can be found in the object code between the addresses $a+3$ and $a+6$, as recorded in the dictionary.

Of course, this method cannot be applied in every case and on all kinds of machine. First of all, code-copying is possible only for *position invariant* instruction sequences. For example, a subexpression that includes a call of a local function by way of a *relative branch* is not position invariant. It happens that information about position invariance can also be recorded conveniently in the dictionary, as shown in the example above.

Secondly, the instruction sequences obtained by code-copying may not be optimal for more complex processors with many registers and multistage instruction-pipelines. For these machines, it may be necessary to employ specific optimization techniques. However, since SDE preserves all information that is available in the source text, arbitrary optimization levels are possible at the time of code generation, albeit without the shortcuts provided by code-copying. One then simply treats the semantic dictionary as an abstract syntax-tree in tabular form.

The interesting part is that code-copying is beneficial especially on machines that are otherwise slow, i.e., CISC processors with few registers. Consider module M again, in which the subexpression $i := i + j$ appears three times. On a simple machine that has only a single accumulator or an expression stack, the identical instruction sequence will have to be used in each occurrence of the subexpression. Code-copying can accelerate the code-generation process in this case. The optimal solution for a modern RISC processor, on the other hand, might well consist of three distinct instruction sequences for the three instances of the subexpression, due to the use of register variables and the effects of instruction pipelining. However, such processors will generally be much faster, counterbalancing the increased code-generation effort, so that an acceptable speed of loading can still be delivered to interactive users relying on dynamic code-generation.

3. Implementation

3.1. Basic Architecture

A *code-generating loader* (CGLoader) has been implemented for an experimental version of MacOberon [Fra90, Fra93a]. MacOberon is a full implementation of the Oberon System [WG89] for Apple Macintosh II [App85] computers and supports *separate compilation* with static interface checking of programs written in the programming language Oberon [Wir88]. It also offers *dynamic loading* of individual modules, meaning that separately compiled modules can be linked into an executing computing session at any time, provided that their interfaces are consistent with the interfaces of the modules that have been loaded already. In MacOberon, dynamic loading is accomplished by a *linking loader*, which modifies the code image retrieved from an object file in such a way that all references to other modules are replaced by absolute addresses.

The new implementation adds a *second loader* to MacOberon, namely a CGLoader processing semantic-dictionary-encoded program files (SDE-files). It has been possible to integrate this CGLoader transparently into the existing MacOberon environment, in the sense that SDE-files and native object-files are completely interchangeable in the implemented system and can, therefore, import from each other arbitrarily. Depending on a *tag* in the file (first two bytes), either the native linking loader or the CGLoader is used to set up the module in memory and prepare it for execution.

The CGLoader has not been incorporated into the core of MacOberon, but constitutes a self-contained module package at the application level. The existing module loader was modified slightly, introducing a *procedure variable* into which an *alternate load procedure* can be installed. Once initialized, this procedure variable is up-called whenever an abnormal tag is detected in an object file, and so initiates the passage of control, from the linking loader that is part of the system core, to the CGLoader external to it. The installation of such an alternate load procedure is considered a privileged operation, analogous to the modification of an interrupt vector. Besides the obvious advantages during the design and testing phases, this architecture permits us to view machine-independence quite naturally as an *enhancement of an existing system*. It also hints at the possibility of adding successive external CGLoaders as time progresses and standards for machine-independent object-file formats emerge. Several such formats could in fact be supported simultaneously, among which one would distinguish by different file-tags.

3.2. Interchangeability of Object Files

SDE-files and their native-object-file counterparts are completely interchangeable in the implemented system. This flexibility doesn't come as readily as it may seem. Not only does it require that the CGLoader is able to interpret *native symbol-files* and the corresponding entry-tables in memory, but also that information can be communicated from the new load architecture to the old one, in formats that are *backward-compatible* with the native compiler and loader.

In the present case, the more difficult part of this problem had an almost trivial solution, since the two varieties of "object" files in the implemented system share a *common representation of symbol-table information*. As documented in an earlier paper [Fra93b], a *portable symbol-file format* had been introduced into MacOberon some time ago, leading to improvements in performance completely unrelated to portability. But now, there were

further benefits from the previous investment into machine-independence: SDE-files use exactly the same symbol-table encoding as the symbol files of the native MacOberon compiler, and the encoded information starts at the same position in both types of file. Consequently, the native MacOberon compiler cannot distinguish SDE-files from its regular symbol-files. It is able to compile modules that import libraries stored in SDE form, as it can extract the required interface information directly from SDE-files.

Apart from the native compiler, the *native loader* need also be able to handle modules that import machine-independent libraries. This is achieved by enabling the CGLoader to construct on-the-fly not only object code, but also all of the remaining data structures that are usually part of a native object-file, such as entry tables. Once that a module has been loaded from an SDE-file, it loses all aspects of portability, and can henceforth be maintained (enumerated, unloaded, etc.) by the regular MacOberon module manager.

3.3. Execution Frequency Hierarchy Considerations

Programs are usually *compiled* far less frequently than they are *executed*. Therefore, it is worthwhile to invest some effort into compilation, because the benefit will be repeated. The same holds for program *loading* versus *execution*. While a program is normally loaded more often than it is compiled, the individual machine instructions are executed even more often, many times on average per load operation. Unfortunately, we need to shift workload unfavourably when we employ a loader that performs code generation. This can be justified only if important advantages are gained in return, or if the resulting code quality is increased. After all, it is execution speed that matters ultimately.

From the outset, three desired properties were therefore put forward that a system incorporating a CGLoader should possess in comparison to a system employing a traditional compiler and linking loader:

1. run-time performance should be at least as good
2. loading time should not be much worse
3. source-to-IL-translation time should be tolerable

These requirements could be met. Not only was it possible to construct a fast CGLoader for *Motorola 68020* processors [Mot87], but the native code produced by it is of high quality. Indeed, although its method of code-generation is quite straightforward, the CGLoader is sometimes able to yield object code that would require a much more sophisticated code-generation strategy in a regular compiler. This is mainly due to the fact that the absolute addresses of all imported objects are known at loading time, which enables the CGLoader to optimize access to them; for example, by using short displacements instead of long ones.

The time required by the CGLoader for creating native code on-the-fly turned out to be quite acceptable, too. Loading with code-generation takes only slightly longer than loading of a native object-file with linking; the difference is barely noticeable in practice. Moreover, the Oberon system has a modular structure, in which many functions are shared between different application packages. In traditional systems, these common functions would be replicated in many applications and statically linked to each of these applications. As a consequence of *modular design*, each new application adds just little code to an already running system. Hence, at most times during normal operation, the CGLoader need only process the moderate number of modules that are unique to an application, while other modules that are also required will already be in memory from past activations of other applications.

4. Results

Four different Oberon application-packages have been used in the benchmarks in this section. *Filler* is a program that draws the Hilbert and Sierpinski varieties of space-filling curves onto the screen. *Hex* is a byte-level file editor of medium sophistication. *Draw* is an extensible object-oriented graphics editor [WG92], of which the three main modules and three extension modules are included in the survey. *Edit* is a relatively sophisticated extensible document processing system [Szy92], five modules of which are studied here.

4.1. Memory and File-Store Requirements

Table 1 gives an impression of the compactness of SDE-files, the influence of the presence of run-time integrity checks on code size, and the memory requirements of the CGLoader. Its first three columns compare the sizes (in bytes) of native MacOberon object-files with those of SDE-files. Two different values are given for the former, reflecting different levels of run-time integrity checking. SDE-files contain enough information to generate code with full integrity checking, but the user need only decide at loading time whether or not he requires code that includes these run-time checks, and which ones should be included.

The column labelled "Native –" in the table below displays the size of native object-files that include reference information for the MacOberon post-mortem debugger, but no extra code for nil-checking, index-checking, type-checking, nor for the initialization of local pointers to NIL. Conversely, the column labelled "Native +" gives the sizes of native object-files that include not only reference information, but also code for the aforementioned run-time checks. Neither kind of object file includes symbolic information for interactive debugging, which can be added by enabling a further option in the MacOberon compiler.

The fourth column of Table 1 shows the maximum size to which the semantic dictionary grows during compilation and code-generation. The CGLoader requires about 80 times this number of bytes as temporary storage while loading a module.

Module	Native –	Native +	SDE	Dictionary
Filler	2900	3236	1232	1003
Hex	10810	12678	4480	1308
Graphics	12692	15935	5281	1382
GraphicFrames	10045	12186	4273	1549
Draw	6033	7133	2193	1498
Rectangles	3255	4297	1378	1271
Curves	5810	7340	2168	1287
Splines	4611	5659	1955	1566
TextFrames	30687	37491	11667	1892
TextPrinter	11503	12537	4494	1486
ParcElems	15259	17540	5273	1437
Edit	11670	13431	5157	1659
EditTools	18756	20898	7397	1488
	144031	170361	56948	

Table 1: Object-File Size (Bytes) and Dictionary Size (Entries)

This data shows that on average, SDE-files are about 2.5 times more compact than native MacOberon object-files not containing run-time integrity checks, and about 3 times more compact than native files incorporating these checks. It is also noteworthy that the maximum size, to which the semantic dictionary grows during encoding and decoding, is not proportional to the overall size of a module, but only roughly proportional to the length of the longest procedure in a module. This seems to level out at a relatively small value in typical modules, much smaller than anticipated originally.

Table 2 indicates how much information is output into memory by the CGLoader. The first column repeats the sizes of SDE-files from the previous table. The second and third columns give two different values for the size of the object code generated on-the-fly, depending on whether or not run-time integrity checks (in the same combinations as before) are emitted. The remaining columns list the sizes of dynamically-generated constant data (including type descriptors), reference data for the Oberon post-mortem debugger, and link data. The latter comprises entry tables generated on-the-fly, so that native client modules can later be connected to dynamically generated library modules.

Module	SDE-File	Code -	Code +	Const	Ref	Link
Filler	1232	2352	2682	139	244	8
Hex	4480	9224	11020	267	1094	12
Graphics	5281	9036	12140	799	1452	204
GraphicFrames	4273	8268	10354	433	824	76
Draw	2193	4754	5752	236	451	60
Rectangles	1378	2644	3596	104	292	20
Curves	2168	5042	6462	100	439	24
Splines	1955	3812	4806	137	437	20
TextFrames	11667	25878	32536	629	2843	156
TextPrinter	4494	9176	10180	478	1367	40
ParcElems	5273	12944	15180	493	1085	52
Edit	5157	9368	10940	555	1098	72
EditTools	7397	14844	16886	723	2009	116
	56948	117342	142534	5093	13635	860

Table 2: Sizes of SDE-Files and of Dynamically-Generated Data (Bytes)

It is notable that SDE-files encode programs more than twice as densely as object code for the MC68020 architecture [Mot87]. If code that includes run-time-checking is considered instead, the factor becomes even 2.5. This is in spite of the fact that SDE-files can additionally also serve as symbol files and contain reference information as well.

4.2. Performance

Unless otherwise noted, the following benchmarks were all carried out under *MacOberon Version 4.03* on a *Macintosh Quadra 840AV* computer (MC68040 Processor running at 40MHz), using version 7.1.1 of the Macintosh System Software. All results are given in real time, i.e. the actual delay that a user experiences sitting in front of a computer, with a resolution of 1/60th of a second. Each benchmark was executed after a cold start, so that no files were left in the operating system's file cache between a compilation and subsequent loading, and the best of three measurements is given in each case. Every attempt has been made to present the system as a regular user would experience it in everyday use.

Table 3 presents the times (in milliseconds) required for compilation and for module loading. The first two columns compare the native compilation times of the MacOberon compiler with the times that the semantic-dictionary encoder requires for generating an SDE-file out of an Oberon source program. The remaining two columns report the loading times for both varieties of object file, including disk access, and, in the case of SDE-files, also including on-the-fly generation of native code. On average, 10% of the object code emitted by the CGLoader can be generated by code-copying. All timings apply to the situation in which no run-time integrity checking is used.

Module	Compile	Encode	Ld Native	Ld SDE
Filler	583	633	83	100
Hex	666	900	100	116
Graphics	766	1083	116	166
GraphicFrames	650	950	66	116
Draw	516	683	66	100
Rectangles	400	500	50	83
Curves	450	650	50	116
Splines	433	583	66	83
TextFrames	1416	2183	216	316
TextPrinter	666	950	100	133
ParcElems	750	1150	116	200
Edit	716	1166	100	166
EditTools	900	1483	150	233
	8912	12914	1279	1928

Table 3: Compilation versus SDE-Encoding and Module Loading Times (ms)

As can be seen from these timings, semantic-dictionary encoding on average takes about 1.5 times as long as normal compilation. I think that there is still room for improvement, as the speed of the encoder was of no major concern during its development. Conversely, I aimed at optimal speed of loading with code generation. This currently takes about 1.5 times as long as normal loading. However the times spent directly on module loading tell only half the story.

Table 4 presents a more realistic measure of loading time, as it takes into account not only the time required for loading an application, but also the duration of *loading and displaying a typical document*. The following timing values indicate how long (in milliseconds) a user must wait after activating a document-opening command before he can execute the next operation. Commands take longer the first time that they are issued, because the modules of the corresponding application have to be loaded as well. Subsequent activations then take up much less time.

Command	Native -	SDE -	Difference
first Draw.Open Counters.Graph	1333	1450	+ 9%
subsequent activations	700	700	
first Edit.Open OberonReport.Text	1400	1766	+ 26 %
subsequent activations	883	883	

Table 4: Command Execution Times with Checks Disabled (ms)

Table 5 demonstrates the effect of run-time integrity-checking on loading time. The measurements of Table 4 have been repeated, but with native object-files that incorporate run-time integrity checks and are therefore larger, and with on-the-fly emission of the same checks in the CGLoader.

Command	Native +	SDE +	Difference
first Draw.Open Counters.Graph	1366	1466	+ 7%
subsequent activations	700	700	
first Edit.Open OberonReport.Text	1550	1800	+ 16%
subsequent activations	916	916	

Table 5: Command Execution Times with Checks Enabled (ms)

The timings in Tables 4 and 5 show that, in practice, on-the-fly code-generation is already almost competitive to dynamic loading of pre-compiled native code from regular object-files. This applies to current state-of-the-art CISC hardware, and is in part due to the fact that disk reading is relatively slow. The compactness of the SDE representation speeds up the disk-access component of program loading considerably, and the time gained thereby counterbalances most of the additional processing necessary for on-the-fly code generation. This argument is supported by a comparison of Tables 4 and 5. It seems to be more efficient to generate run-time checks on the fly than to inflate the size of object files by including them there.

A noteworthy trend in hardware technology today is that processor power is rising more rapidly than disk access times and transfer rates. This trend is likely to continue in the future, which means that hardware technology is evolving in favor of the ideas proposed here. Consider Tables 6 and 7, which repeat the timings of Table 5 for different processors of the MC680x0 family, using the identical external disk drive.

Machine	Native +	SDE +	Difference
Macintosh II (16MHz MC68020)	5683	7966	+ 40%
Macintosh IIx (16MHz MC68030)	4300	5933	+ 38%
Macintosh IIfx (40MHz MC68030)	1833	2283	+ 25%
Macintosh Quadra 840AV (40MHz MC68040)	1366	1466	+ 7%

Table 6: Time for *Draw.Open Counters.Graph* on Different Machines (ms)

Machine	Native +	SDE +	Difference
Macintosh II (16MHz MC68020)	4733	8850	+ 87%
Macintosh IIx (16MHz MC68030)	3600	6750	+ 88%
Macintosh IIfx (40MHz MC68030)	1550	2400	+ 55%
Macintosh Quadra 840AV (40MHz MC68040)	1550	1800	+ 16%

Table 7: Time for *Edit.Open OberonReport.Text* on Different Machines (ms)

Extrapolating from these results, there is reason to believe that on-the-fly code-generation from small SDE-files may eventually become faster than dynamic loading of larger native object-files, unless of course secondary storage universally migrates to a much faster technology. One way or the other, on-the-fly code-generation will definitely become faster in absolute terms as clock speeds increase further, so that the relative speed in comparison to native loading should not much longer be of importance anyway. Ultimately, the speed of loading needs to be tolerable for an interactive user – that is all that matters.

4.3. Code Quality

The last point that needs to be addressed concerns the quality of code that is generated dynamically. Table 8, by way of a popular benchmark, compares the quality of code generated on-the-fly with that generated by the *Apple MPW C compiler for the Macintosh (Version 3.2.4)* with all possible optimizations for speed enabled ("*-m -mc68020 -mc68881 -opt full -opt speed*"). The generation of integrity checks was disabled in the CGLoader, because no equivalent concept is present in *C*.

The table lists execution times in milliseconds (less is better). Due to processor cache effects, these timings can vary by as much as 15% when executed repeatedly; the figures give the best of three executions. The *C* version of the *Treesort* benchmark exceeded all meaningful time bounds, due to apparent limitations of the standard Macintosh operating system storage-allocator. The CGLoader is not bound by these limitations, as it generates calls to the storage-allocator of *MacOberon*, which includes an independent memory-management subsystem.

Benchmark	SDE -	MPW C
Permutation	83	113
Towers of Hanoi	83	121
Eight Queens	50	43
Integer Matrix Multiplication	150	173
Real Matrix Multiplication	133	171
Puzzle	800	800
Quicksort	66	61
Bubblesort	117	88
Treesort	83	> 1000
Fast Fourier Transform	133	123

Table 8: Benchmark Execution Times (ms)

From this data, it can be inferred that the CGLoader emits native code of high quality that can compete with optimizing *C* compilers. In some cases, it surpasses even the output of the official optimizing compiler recommended by the manufacturer of the target machine, which is orders of magnitude slower in compilation. It should be possible to improve the remaining cases in which the CGLoader is currently inferior, without sacrificing much of the speed of on-the-fly code generation. Since the efficiency of the CGLoader is rooted in the compactness of the abstract program representation, rather than in any machine-specific details, it should be possible also to duplicate it for other architectures.

The *Apple MPW C* compiler requires about 6.9 seconds for compiling the *C* source program for the benchmark, and a similar time additionally for linking, compared to 1.1 seconds that are needed for encoding the corresponding Oberon program into an SDE-file and 233 milliseconds for loading it with on-the-fly code generation (including file access after a cold start).

5. Applications

5.1. Software Components

At the 1968 NATO conference, McIlroy [McI68] argued that "software production today appears in the scale of industrialization somewhere below the more backward construction industries" and attributed this to the absence of a *software component industry*. Yet more than twenty-five years later, not much has changed in the way we construct software. Reuse of software at best takes place within commercial organizations, but not between them. An industrial programmer still cannot just open a catalog of standard software parts and order a module from it that will execute an algorithm according to some given specifications. At best, he can hope that the algorithm he requires is built into the operating system. Strangely enough, today we witness an ongoing standardization of software functions by incorporation into operating systems and related libraries, instead of a separate industry developing software components.

Why, then, has no independent market for platform-independent standard software components developed over the years? Why have operating systems and their supporting libraries instead grown to an awesome complexity, encompassing functions as diverse as user-interface management and data-base support? The answer may simply be that there is currently no commercial incentive to develop *plug-in software components* because the maintenance costs would be prohibitive in today's marketplace. An independent software-component vendor would have to provide his products either in a multitude of link and object formats, which is costly, or in source form, which requires complex legal arrangements to protect the intellectual property of the authors and might cost even more. On the other hand, there is a direct commercial advantage for an operating-system vendor when he adds functionality to his product. As a consequence, we see a proliferation of operating-system-level enhancements instead of an intermediate industry for operating-system-independent support libraries.

The work presented in this paper contributes to lowering the cost of providing drop-in software components. It demonstrates the feasibility of a platform-independent software distribution format that enables portable modules to be used right out-of-the-box, without any off-line steps of compilation or linking. The *on-line* aspect is important because it means that even end-users can migrate libraries in their possession to new hardware platforms. It also suggests that different component-vendors could offer competing implementations of the same library, which an end-user would install or replace simply by *plugging in*. This is analogous to the situation in the personal computer hardware market, in which end-users are expected to buy and install themselves certain parts such as floating-point coprocessors.

5.2. Run-Time Integrity Checks

Apart from simplifying the distribution of software modules that are to be used on different target architectures, the proposed scheme eliminates also many of the cases that traditionally require *several versions* of a module to coexist side-by-side on a single target machine. During development, we often require variants of standard library modules because the software being developed might not be robust enough to guarantee that all constraints of the library are fulfilled. Accordingly, a *development version* includes additional checks that validate the arguments passed to library routines. For reasons of efficiency,

however, one would not want to perform these validations during regular operation when all clients of the library have been thoroughly tested and can be trusted. Consequently, the tests are usually removed from the *production version*.

For example, consider a procedure *ReadBytes*(*f*: *File*; *VAR b*: *ARRAY OF CHAR*; *n*: *LONOINT*) in module *Files*. It is essential that the procedure never reads beyond the end of the input buffer, i.e. the precondition $n \leq \text{LEN}(b)$ must hold. In some cases, the compiler alone may be able to verify that this condition is satisfied, given that there is a suitable mechanism for specifying such context requirements. However, there will always be cases in which the precondition can only be verified at run-time. Therefore, a corresponding validation needs to be present in the development version of module *Files*, but not necessarily in the production version. Unfortunately, a serious management problem arises from having to maintain more than one version of the same module, and having to keep track of which one is which. One constantly has to make sure that changes in a source text are propagated to all variants of the module that may exist concurrently, and use an elaborate naming scheme to differentiate between compiled variants of the same module.

On-the-fly code generation does away with module variants and the associated management overhead. Only at the time of loading do we need to indicate whether we require a *development* or a *production* version of a module. The implemented system supports several independent run-time integrity guards that can be enabled selectively (Table 9). Instead of the individual object file, it is then the run-time-environment that dictates whether or not these guards will be generated. Likewise, symbolic reference information for a debugger can be created as needed, and need not take up disk and memory space in configurations that are used solely for running finished applications. For example, the current implementation provides for the optional insertion of routine names in the code, in a format acceptable to the standard debugger of the host machine.

Switch	Effect if Enabled
debug	annotate code for symbolic debugging
clear	initialize local pointer variables to NIL at procedure entry
nil	test pointers for NIL prior to dereferencing
index	check that array subscripts lie within bounds
type	perform run-time type tests (used with type extension)
assert	generate code for ASSERT function

Table 9: Code Generation Switches

The key to *argument validation* in the implemented system lies in the standard function *ASSERT* of the programming language Oberon [Wir88]. *ASSERT* accepts as its arguments a Boolean expression and an Integer constant. It has the effect of a run-time test to check if the Boolean expression yields *TRUE*. If it doesn't, a run-time trap to the exception vector indicated by the second argument is taken. The main point about *ASSERT* is that it can be turned off by a compiler option, so that no code will be generated at all. Allowing the user to decide at run-time whether or not assertions should be verified eliminates the need for a separate development environment.

5.3. Management of Changes

Having available a system in which native code is generated only at the time of loading reduces also the organizational overhead required to keep a modular application

consistent. Each time that a module is loaded, the implemented system performs a recompilation of the module's implementation, but in a manner that is completely transparent to the user. Consequently, the effects of certain changes of library modules can remain invisible, in contrast to other systems in which source-level recompilations of client modules are unavoidable.

A module needs to be recompiled whenever its own implementation is changed, or whenever it is invalidated by a change in one of the modules it depends on. Deciding which clients are invalidated by a change in a library is the difficult part. The easiest solution, implemented in tools such as the *Make* utility [Fel79] of the UNIX operating system [TR74], is to invalidate *all clients* each time that a library is changed. However, this introduces many recompilations that could be avoided in principle.

Tichy and Baker [TB85, Tic86] have introduced the notion of *smart recompilation*. This technique is founded on a detailed analysis of import/export relationships, considering not only the involved modules as a whole, but also the individual features that are exported from one module and imported by another. The results of this analysis are maintained in a database, along with a module dependency graph. One can then mechanize the decision which modules need to be recompiled by comparing the *changed feature set* of a modified library with the *referenced feature sets* of clients. In some cases, such as the addition of further procedures to a library, an interface change need not invalidate all existing clients then.

The proposed method opens a path to even further reduce the number of (source-text) recompilations. A large proportion of changes that typically occur during the software life-cycle has no effect on the *behavior* of a program but nevertheless on the machine code being generated, because native code contains addresses, sizes, and relative offsets *literally*, and addressing modes often depend on particular address values. An example for a change that has consequences only in the code generator is the *insertion of an additional data field* in an exported record type. Apart from the possible error that may occur if the corresponding identifier is used already within the scope of the record, a condition that is detected easily, this addition preserves the semantics of all client modules. Unfortunately, however, the addition alters the size of the record type, and may change the relative offsets of some of the existing record fields, so that new code needs to be generated for all modules that use the record type. In a system such as the implemented one, code generation happens transparently and need not be of concern to users, because all client modules will be updated automatically when they are loaded the next time.

In principle, therefore, in a system in which code generation occurs at loading time, recompilation (of source code) is *not required to propagate changes*. There are, of course, certain changes that invalidate the source code of some client modules completely, such as removing a routine from a library module or changing the result type of a library function, but these require some *source-level re-coding* of all affected clients and cannot simply be dealt with by simple recompilation. Such situations can be detected in advance using the techniques described by Tichy and Baker, or will in any event be flagged when the CGLoader senses an interface mismatch, resulting in a load error.

Accordingly, in a system offering on-the-fly code generation at load time, the only factor that determines whether (source-text) recompilation of clients is necessary in reaction to changes in libraries, is the strategy that is used for describing the inter-module links in the symbol table. The current implementation uses *per-module fingerprints* to ensure interface consistency, so that more recompilations are needed than would be necessary if *per-object fingerprints* were used. In the latter case, no recompilations of clients would be necessary

ever. However, this aspect has not been the main focus of the current work and, therefore, not been pursued.

5.4. Improving Code Quality by Targeted Optimizations

The fact that object code is generated anew each time that a module is loaded could also be exploited for increasing the *overall performance* of the whole system, although this has currently not been implemented. Borrowing from ideas discussed by Morris [Mor91] and Wall [Wal91], an *execution profile* obtained in a previous run of the system could guide the level of optimization applied by the CGLoader in the creation of the next executable version.

While fine-grained profiling might be useful as a basis for specifically-targeted optimizations, run-time profiling, which is associated with an overhead, might not even be necessary in a modular system. Instead, one might simply use the *import counter*, which indicates how many clients a module has, as an estimate for the "relative importance" of a module. The more important a module is, the greater the potential benefits of optimization.

The idea of targeted optimization may be developed even further, taking into account that object code can be re-created from SDE-files at any time. The system might expend its *idle time* on the recompilation of whole subtrees of the loaded-module graph, employing a higher optimization level than the currently loaded version. After recompiling such a subtree, it may then attempt to unload the old version, and if unloading is successful adjust the global module graph to include the new, optimized version. Note that the unloading step may fail if further clients are added to the originally loaded modules while re-generation is underway, or if installed procedures from the original modules remain active in the system.

5.5. Further Applications

A machine-independent abstract program representation from which high-quality code can efficiently be generated on-the-fly might also prove to be valuable in the context of heterogeneous distributed systems consisting of several different hardware platforms.

For example, the proposed technique could form the basis for a very general *remote procedure-call* mechanism [Nel81]. Instead of compiling a separate stub for each procedure to be called remotely and installing it as a process on the target machine, one might send a complete instruction sequence in a machine-independent format, to be processed by a single *code-generating stub* on the side of the receiver. Cryptological authentication measures could be applied to prevent misuse in an open network.

Consistency problems between program segments for different machines in a distributed application could be avoided trivially by sending a *consistent version* across the network before the start of the distributed computation, thereby guaranteeing that identical code executes on all machines taking part in the computation.

Last but not least, designers of object-oriented systems could use an even broader definition of *object persistence*. A persistent object might contain its own code in an abstract format. Such an object could then migrate over a network or be transported on some storage medium. At a destination site, first the code to handle the object would be generated dynamically. Thereafter, the object's data would be read in.

6. Comparison with Related Work

The project described here was started in 1990, and first results were published in September of 1991 [FL91]. At that time, it seemed almost exotic to attempt any revival of the old *UNCOL* idea. Today, the topic again seems "hot", and many researchers are working on related projects. The most notable of these other projects is the *architecture neutral distribution format (ANDF)* initiative by the *Open Software Foundation (OSF)*.

6.1. The OSF ANDF Project

The Open Software Foundation (OSF) has recently adopted a technology called *TDF* [OSF91, DRA93a, DRA93b], designed and implemented by the United Kingdom Defence Research Agency (DRA), to serve as the basis of an *architecture neutral software distribution format (ANDF)*. TDF has many characteristics in common with semantic-dictionary encoding. Just like SDE, and unlike previous *UNCOL* attempts, TDF is not based on an abstract machine, but on a tree-structured intermediate language in conjunction with an embedded symbol table.

TDF has been designed to be both source-language and target-architecture independent, although as of June 1993, the only existing compiler front-end for TDF was for the programming language *C* [KR78]. TDF is claimed to be useful for source languages other than *C*, and compilers translating into the TDF representation from other languages are being developed. However, although TDF is extensible to accommodate specific features of future programming languages, it is not guaranteed that this can be done in an upward-compatible manner [DRA93c].

In contrast, the method of SDE is not a program representation in its own right, but a *meta-technique* for encoding programs abstractly. It is parametrized by the initial configuration of the dictionary and the heuristics used for dictionary management. So far, SDE has been applied only to programs originally written in the programming language *Oberon* [Wir88]. However, since semantic meaning is instilled into each SDE-file solely by the *initial configuration* of a dictionary, it is possible to support easily future language requirements, even without invalidating existing SDE-files in an old format. All that is necessary is a key in the SDE-file that uniquely identifies the initial configuration of the dictionary that has to be used for decoding. This might, for example, be simply the name of a file in which the configuration is stored.

Hence, SDE allows us to *evolve the set of encodeable language constructs* independently of the actual file-formats. In fact, by using configuration files, individual SDE-decoders could be made completely independent of the file formats they need to process. It would require a standardization only of the *meanings* of different meta-language constructs. Software developers would then be able to choose freely which of these meta-language constructs to use in the encoding of their programs, and at which positions of the initial dictionary they would place these constructs. The smaller the set of meta-language constructs, the more difficult it will be to reverse-engineer the encoded program, although the use of fewer constructs could also affect compactness and optimizeability adversely.

The question of reverse-engineerability is important for software developers wishing to offer their products in a portable format. SDE and TDF both preserve the abstract structure of programs and can, therefore, be reverse-engineered to produce a "shrouded" source program, i.e. one that contains no meaningful internal identifiers [Mac93]. However, with current technology, reverse-engineering to a similar degree is possible also from binary

code. Many of the algorithms that have been developed for object-code-level optimization [DF84] are useful for these purposes. Moreover, the statement [DRA93c] is probably correct that portable formats are such attractive targets to reverse-engineer that suitable tools will become available anyway, regardless of how difficult it is to produce such tools. It would, therefore, not make much sense to jeopardize the advantages of SDE in an attempt to make reverse-engineering more difficult.

SDE has some further advantages over TDF. To start with, it is more compact. Although in [OSF91] the OSF recognized that "*it is important that the ANDF file size be as small as possible*", TDF is in fact less compact than object code. TDF is quoted [DRA93c] as being "around twice the size of the binary of CISC machines", while SDE is at most half the size of MC68020 binary code. Because of the lack of a common operating platform on which both mechanisms have been implemented, a direct performance comparison of SDE versus TDF is currently not possible. However, DRA [DRA93c] state that native object-file generation, which is an off-line process in their implementation, takes between 32% and 83% of native C compile time. Considering that Oberon compilers usually outperform C compilers by a factor of more than 15 in compilation speed [BCF92], and that loading of SDE-files is more than four times faster than regular Oberon compilation, it seems reasonable to claim that, at least on CISC processors and without taking code quality into account, SDE should provide for much faster code generation than TDF.

6.2. Dynamic Code-Generation

The concept of *dynamic translation* of programs from one representation into another has been around for some time. Early implementations, such as one by Brown [Bro76], were developed with the aim of balancing execution speed and memory requirements under the extreme hardware constraints that were then the norm. These early implementations applied only to programming languages without block-structure and performed the generation of native object-code on a statement-by-statement basis.

For a long time, the main reason for implementing dynamic translation remained the fact that it allowed an elegant trade-off between execution efficiency and memory consumption. A paper by Rau [Rau78] classifies program representations into three categories, namely *high-level*, *directly interpretable*, and *directly executable*, and discusses the use of dynamic translation between these categories as a means for achieving speed and compactness simultaneously.

Then came Deutsch and Schiffmann's [DS84] landmark paper on the efficient implementation of the programming language *Smalltalk-80* [GR83]. They used dynamic translation for increasing execution speed while *retaining virtual-machine code-compatibility* with existing implementations. The latter was necessary because the Smalltalk-80 virtual machine is actually visible to user programs, and much of the system code depends on it. In Deutsch and Schiffmann's implementation, native code for the actual target machine is generated on-the-fly and cached until it is invalidated by changes in the source program, or until it is overwritten in the code-cache due to lack of space.

A more recent application of dynamic translation comes from the implementation [CUL89, CU89] of the programming language *Self* [US87], a dynamically-typed language based on prototypes. Just as the Smalltalk-80 system, it can benefit enormously from on-the-fly code generation, because type information, although unavailable statically, is available at run-time. By allowing the dynamic generation of several variants of an expression, optimized for different run-time types of the component variables, the

efficiency of such systems can be multiplied, but still cannot compete with statically-typed programming languages.

In contrast, the implemented system attempts to deliver run-time performance comparable to traditional compilers and offers on-the-fly code generation primarily as a means for increased user convenience, not code quality. It ties dynamic translation intimately to the two concepts of *separate compilation* and *dynamic module loading*. The information contained in SDE-files is equivalent to the source description, so that there is no fundamental limit to the obtainable code quality. Hence, the most effective optimizing code generators could potentially be built into a CGLoader operating on SDE.

It is also true that *modules* are much better suited as the unit of code generation than procedures. A module is a collection of data types, variables, and procedures that are loaded together always, and, almost equally important, unloaded together always. Furthermore, a module can be loaded only after all of its servers (imported library modules) have been loaded successfully, and unloaded only after all of its clients (importing modules) have been unloaded. Consequently, code is generated from the bottom upwards, and the addresses of all callees are known when compiling a caller. Likewise, there are never any clients that need to be invalidated explicitly when a module is unloaded.

In systems in which the unit of code generation is the *procedure*, such as Smalltalk-80 and Self, program execution is interspersed with code-generation. Each procedure call may potentially fault, at which point native code needs to be generated dynamically before the call can be completed. Unfortunately, this may sometimes generate formidable amounts of such faults in succession, each associated with a re-load of the instruction cache, causing disruptive delays for interactive users.

6.3. Dynamic Binary-To-Binary Object-Code Translation

A technology that has emerged only recently is *binary translation* of object code [SCK93]. It enables programs for one architecture to be executed directly on another via true translation of whole object programs into the native instruction set of the new target machine. The main rationale behind binary translation is the need to protect previous investments into software when migrating to a new hardware architecture. Rather than constituting a *portability technique* in the spirit of "software components", binary translation represents a *capitulation* before the fact that much of the software in existence is not portable, and cannot be ported by ordinary means; for example, because the source texts and the original design documents are no longer available.

Binary translation is a complex technique. Since it is put to use mainly in circumstances in which little is known about the programs that serve as its input, the translation mechanism needs to be suitable for any program that could possibly have executed on the architecture of origin. This includes programs that modify themselves. Consequently, self-modification conditions need to be detected on the new target machine, at which time the affected code segments may have to be re-translated on-the-fly.

Due to its complexity, binary translation cannot really be seen as an answer to the portability problem, but only as an intermediate solution allowing us to keep using an existing software base while it is being rewritten for the new architecture. The technique that I am advocating in this paper is orders of magnitude simpler and concerned less with backward-compatibility than with forward-compatibility, with whatever may lie ahead in the future.

7. Future Work

As a next step, a second CGLoader for a different architecture is planned that will operate on the identical SDE-file format. This would provide object-level portability between different processor architectures. In fact, it should be noted that traditional "binary" compatibility may not be good enough for future machines anyway. This is because different implementations of the same architecture begin to diverge by so much that it is becoming virtually impossible to generate native object-code that will perform well on all processors within a family. Among other features, different implementations of an architecture stand apart in the number of instructions that can be issued simultaneously to independent functional units, and in the depth of their instruction pipelines. On the other hand, the use of a CGLoader allows for the scheduling of instructions specifically for each particular target processor. It is thereby possible to deliver object code that is custom-tailored towards each processor's characteristics, along with a user convenience comparable to that of binary compatibility.

8. Summary and Conclusion

This paper has described a new technique for representing programs abstractly, which yields a highly compact encoding and is able to provide a code generator with all the information available on the level of the source language, plus additional knowledge about the occurrence of common subexpressions in the source text. It facilitates simple and efficient on-the-fly code generation on relatively slow processors without precluding the use of highly optimizing code generation methods on faster ones.

The new technique is able to provide separate compilation of program modules with type-safe interfaces, independent module distribution, and interchangeability of modules with the same interface. These properties give it an advantage over other approaches to portability, such as using the programming language C [KR78]. It is also more practical than "shrouded" C [Mac93], which might be seen as a sort of portable assembly-language that is yet difficult to reverse-engineer. While it is true that many of the cost-lowering benefits of portability can be gained by using obscured C, or any high-level language for that matter, code generation from the SDE representation is orders of magnitude faster than the compilation of a C program source.

The proposed technique offers the potential of providing a universal software distribution format that is practical to use. Hence, it might encourage software developers to share reusable software components or offer them commercially. It eliminates the need for separate development environments and can reduce the number of recompilations after changes in library modules. Other potential applications of object-level portability may lie in heterogeneous distributed computing environments, in which the compactness of the SDE representation would be of particular advantage when network transfer is required.

Acknowledgments

The author is indebted to Niklaus Wirth for his guidance of the project described here, and for his numerous suggestions and comments. He would like to thank Jürg Gutknecht, who commented thoroughly on this paper and provided valuable criticisms that helped to improve the presentation of this material considerably.

References

- [App85] Apple Computer, Inc.; *Inside Macintosh*; Addison-Wesley; 1985ff.
- [ADH89] R. Atkinson, A. Demers, C. Hauser, Ch. Jacobi, P. Kessler and M. Weiser; Experiences Creating a Portable Cedar; *Proc. Sigplan '89 Conf. Programming Language Design and Implementation*, published as *Sigplan Notices*, 24:7, 322–329; 1989.
- [BCF92] M. Brandis, R. Crelier, M. Franz and J. Templ; *The Oberon System Family*; Report #174, Departement Informatik, ETH Zurich; 1992.
- [Bro76] P. J. Brown; Throw-Away Compiling; *Software-Practice and Experience*, 6:3, 423–434; 1972.
- [CU89] C. Chambers and D. Ungar; Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language; *Proc. ACM Sigplan '89 Conf. Programming Language Design and Implementation*, published as *Sigplan Notices*, 24:7, 146–160; 1989.
- [CUL89] C. Chambers, D. Ungar and E. Lee; An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes; *OOPSLA '89 Conf. Proc.*, published as *Sigplan Notices*, 24:10, 49–70; 1989.
- [Con58] M. E. Conway; Proposal for an UNCOL; *Comm. ACM*, 1:10 5–8; 1958.
- [DF84] J. W. Davidson and C. W. Fraser; Code Selection through Object Code Optimization; *ACM Trans. Programming Languages and Systems*, 6:4, 505–526; 1984.
- [DRA93a] United Kingdom Defence Research Agency; *TDF Specification, Issue 2.1*; June 1993.
- [DRA93b] United Kingdom Defence Research Agency; *A Guide to the TDF Specification, Issue 2.1.0*; June 1993.
- [DRA93c] United Kingdom Defence Research Agency; *Frequently Asked Questions about ANDF, Issue 1.1*; June 1993.
- [DS84] L. P. Deutsch and A. M. Schiffmann; Efficient Implementation of the Smalltalk-80 System; *Conf. Record 11th Annual ACM Symp. Principles of Programming Languages*, Salt Lake City, Utah, 297–302; 1984.
- [Fel79] S. I. Feldman; Make: A Program for Maintaining Computer Programs; *Software-Practice and Experience*, 9:4, 255–265; 1979.
- [Fra90] M. Franz; *The Implementation of MacOberon*; Report #141, Departement Informatik, ETH Zürich; 1990.
- [Fra93a] M. Franz; Emulating an Operating System on Top of Another; *Software-Practice and Experience*, 23:6, 677–692; June 1993.
- [Fra93b] M. Franz; The Case for Universal Symbol Files; *Structured Programming*, 14:3, 136–147; October 1993.
- [FL91] M. Franz and S. Ludwig; Portability Redefined; *Proc. 2nd Int. Modula-2 Conf.*, Loughborough, England; 1991.
- [GF84] M. Ganapathi and C. N. Fischer; Attributed Linear Intermediate Representations for Retargetable Code Generators; *Software-Practice and Experience*, 14:4, 347–364; 1984.
- [GR83] A. Goldberg and D. Robson; *Smalltalk-80: The Language and its Implementation*; Addison-Wesley; 1983.
- [Gri72] R. E. Griswold; *The Macro Implementation of SNOBOL4: A Case Study in Machine-Independent Software Development*; Freeman, San Francisco, 1972.
- [KR78] B. W. Kernighan and D. M. Ritchie; *The C Programming Language*; Prentice-Hall; 1978.
- [Mac93] S. Macrakis; *Protecting Source Code with ANDF*; Open Software Foundation Research Institute; June 1993.
- [McI68] M. D. McIlroy; Mass Produced Software Components; in Naur, Randell, Buxton (eds.), *Software Engineering: Concepts and Techniques*, Proceedings of the NATO Conferences, New York, 88–98; 1976.
- [Mor91] W. G. Morris; CCG: A Prototype Coagulating Code Generator; *Proc. ACM Sigplan '91 Conf. Programming Language Design and Implementation*, published as *Sigplan Notices*, 26:6, 45–58; 1991.
- [Mot87] Motorola, Inc.; *M68030 Enhanced 32-bit Microprocessor User's Manual*; Motorola Customer Order No. MC68020UM/AD; 1987.
- [Nel81] B. J. Nelson; *Remote Procedure Call*; Report #CLS-81-9, Palo Alto Research Center, Xerox Corporation, Palo Alto, California; 1981.
- [NAJ76] K. V. Nori, U. Amman, K. Jensen, H. H. Nägeli and Ch. Jacobi; Pascal-P Implementation Notes; in D.W. Barron, editor; *Pascal: The Language and its Implementation*; Wiley, Chichester; 1981.
- [OSF91] Open Software Foundation; *OSF Architecture-Neutral Distribution Format Rationale*; 1991.
- [PW69] P. C. Poole and W. M. Waite; Machine Independent Software; *Proc. ACM 2nd Symp. Operating System Principles*; Princeton, New Jersey; 1969.
- [Rau78] B. R. Rau; Levels of Representation of Programs and the Architecture of Universal Host Machines; *Proc. 11th Annual Microprogramming Workshop*, Pacific Grove, California, 67–79; 1978.
- [Rei89] M. Reiser; Private Communication; 1989.
- [Ric71] M. Richards; The Portability of the BCPL Compiler; *Software-Practice and Experience*, 1:2, 135–146; 1971.

- [SCK93] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks and S. G. Robinson; Binary Translation; *Comm. ACM*, 36:2, 69–81; February 1993.
- [SWT58] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock and T. B. Steel; The Problem of Programming Communication with Changing Machines: A Proposed Solution: Report of the Share Ad-Hoc Committee on Universal Languages; *Comm. ACM*, 1:8, 12–18, and 1:9, 9–15; 1958.
- [Szy92] C. A. Szyperski; Write-ing Applications: Designing an Extensible Text Editor as an Application Framework; *Proc. 7th Int. Conf. Technology of Object-Oriented Languages and Systems (TOOLS92)*, Dortmund, Germany, 247–261; 1992.
- [TR74] K. Thompson and D. M. Ritchie; The UNIX Time-Sharing System; *Comm. ACM*, 17:2, 1931–1946; 1974.
- [Tic86] W. F. Tichy; Smart Recompilation; *ACM Trans. Programming Languages and Systems*, 8:3, 273–291; 1986.
- [TB85] W. F. Tichy and M. C. Baker; Smart Recompilation; *Conf. Record 12th Annual ACM Symp. Principles of Programming Languages*, New Orleans, Louisiana, 236–244; 1985.
- [US87] D. Ungar and R. B. Smith; Self: The Power of Simplicity; *OOPSLA '87 Conf. Proc.*, published as *Sigplan Notices*, 22:12, 227–242; 1987.
- [Wal91] D. W. Wall; Predicting Program Behavior Using Real or Estimated Profiles; *Proc. ACM Sigplan '91 Conf. Programming Language Design and Implementation*, published as *Sigplan Notices*, 26:6, 59–70; 1991.
- [Wel84] T. A. Welch; A Technique for High-Performance Data Compression; *IEEE Computer*, 17:6, 8–19; 1984.
- [Wir88] N. Wirth; The Programming Language Oberon; *Software—Practice and Experience*, 18:7, 671–690; 1988.
- [WG89] N. Wirth and J. Gutknecht; The Oberon System; *Software—Practice and Experience*, 19:9, 857–893; 1989.
- [WG92] N. Wirth and J. Gutknecht; *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley; 1992.