Technische Universität Berlin

Forschungsberichte
des Fachbereichs Informatik

A Pragmatic Approach to Software Documentation

Klaus Didrich
Torsten Klein

Report No. 96–4
November 1996

Technische Universität Berlin
Fachbereich 13 • Informatik
Franklinstraße 28/29 • D - 10587 Berlin

A Pragmatic Approach to Software Documentation

Klaus Didrich* Torsten Klein†

Abstract

We present an approach for designing a literate programming tool that, in addition to covering the technical issues, especially targets the acceptance of the documentation system by the program developer. We then describe the DosfOp documentation system for the Opal language, which was developed in line with these design principles. Finally, we discuss experiences with the system from a user's and from an implementor's point of view.

1 Introduction

The need to document software products as soon as they have evolved beyond the stage of mere playthings or examples is evident to most software users and even to most software developers. Nevertheless, documentation is often not available or is outdated, either because the development of actual running software is more important and is easier to check for the customer than the quality of the documentation or because the job of keeping the documentation up to date is too arduous.

Documentation systems can help to combat the second problem. There are now several systems available, starting with Knuth's original `web`[1] [Knu83]. We will discuss some of them in the following section. Their success is, however, only limited. It seems that one needs discipline and a strong faith in the usefulness of software documentation to use such a system. We would not go as far as C. van Wyk, who writes that "[a] fair conclusion from my mail would be that one must write one's own system before one can write a literate program" [Van90], but clearly it is not easy to persuade people to use a system for documenting software.

If we are to propose yet another approach to software documentation we have to be careful about the requirements for such systems. In this regard, the remarks by Will Partain on the creation of the Glasgow literate programming system

*TU Berlin, FB Informatik, Group Compiler Construction and Programming Languages
†TU Berlin, FB Informatik; now at Daimler-Benz AG Forschung Systemtechnik Berlin

[1] ... `web` has nothing in common with the popular Internet communication concept "World Wide Web".

[GRA92], describing not only the system itself but also its development and origins and experiences with certain of its features, have been very helpful.

The purpose of this paper is to present our approach, and to show by experiment that a documentation system that is written according to the principles of this approach is actually not only used by its authors but also by other developers.

The documentation system we have developed is called DosfOp. It began as a documentation system for the functional programming language Opal [DFG+94], which was developed at the TU Berlin by our group. While some features are specific to Opal, or more generally to functional programming languages, the main objectives of the documentation system are language independent. We use DosfOp to illustrate our approach to documentation system development.

- In software engineering, modules are used to structure the software system. These modules form a hierarchy reflecting the logical relations between modules or groups of modules. We expect a documentation system to support the documentation of a software product in a way that reflects its structure.
- The documentation system should use the information that is already contained in the sources. Some people say that well-written programs — with self-explanatory variable and function names — need not be documented at all. We do not subscribe to this extreme opinion, but do believe that a documentation system should include indices, reference tables and the like that refer directly to the elements of the source code.
- The documentation should not only be available in print but also online. Paper is good for documenting static versions of a software product, but we also need the support of a documentation system in the dynamic stages of a development.

In addition to the technical requirements, we also want to take human idiosyncrasies into account:

- Most programmers are very reluctant to use a system if they feel their individuality is not given consideration. So the documentation system should incorporate a lot of bells and whistles enabling the user to customize the outcome.[2]
- The initial effort required to use the documentation system must be very small. Ideally, the user would provide documentation information in the proper places and the system would generate the documentation without any extra activity by the user.

[2]The three people to first use DosfOp each had their own opinion as to how structures should be ordered.

- In particular, it must be possible to integrate source code that was not prepared specifically for the documentation system. So quick-and-dirty programs (which is the way most programs are born) can be integrated and then later on be gradually documented. This is certainly not a very pure approach, but we do think that it leads to better documented source code.

A first prototype of the system was developed by one of the authors (Torsten Klein) as his *Studienarbeit* – undergraduate dissertation – and further enhanced with user-friendly aspects as part of his master's thesis [Kle95], both of which were supervised by the other author (Klaus Didrich). (So what you are reading now is a mixture of an author's description and an evaluation by one (biased) customer.) The system was developed in OPAL itself, with the exception of the graphical user interface, which was developed in TCL/TK. DOSFOP is supplied with the OPAL distribution [Opa95].

2 Literate Programming Systems

In the following section we try to place our research work in the context of existing literate programming concepts and tools. We will deal with the reasons why literate programming is not in the main focus of interest for software developers today and will discuss what we think should be done to combat the general aversion against the documentation of source code.

2.1 Benefits and Drawbacks of Existing Concepts and Tools

In introducing the `web` concepts in 1984, Donald E. Knuth [Knu84] tried to initiate a new era of so-called "Literate Programming"; this was ultimately expected to yield better software that is even "fun to read". The research and improvements on the `web` ideas have primarily been carried out in recent years by tool developers, all of them aiming at reinforcing or removing the emphasis on certain aspects of the original `web` designed for PASCAL.

The concept of the original PASCAL `web` and all its derivatives is based on the two filters, "weave" and "tangle", which perform specific transformations based on a common `web` input file consisting of code and informal documentation. The "weave" filter is designed for the production of pretty-printed documentation intended for human reading. The succession of code and text is identical to the original `web` input file and not restricted by compiler requirements such as "definition before use". The "tangle" filter, on the other hand, deals with these compiler requirements and performs the code-reordering and macro-substitution needed for further automatic processing. Informal documentation is fully dispensed with because it has no effect on the running program.

Much effort has been invested in adapting the `web` idea

- to different programming languages (e.g. FORTRAN [AO90], C [KL93]),
- such that the `web` tool can be tailored to a specific programming language (Spider `web` [Ram89]),
- such that `web` can remain language independent (`noweb` [Ram92]).

As we see it, all derivatives of the original PASCAL `web` show inadequacies when it comes to the solution of some important *conceptional* documentation requirements:

- Writing programs with a `web` system is more than just writing programs in your favourite language with a little bit of added documentation. All `web` derivatives provide a *new* (meta-)language defined on the basis of a programming language and a language intended for typesetting (such as TEX or `troff`). Additionally, this new language introduces some new principles into (i.e. removes some deficiencies in) the underlying programming language.

  For software developers this is an *obstacle* that is difficult to overcome because they have to familiarize themselves with a new formal language and its environment (error messages, debugging techniques, etc.) and have to *rewrite existing code* so that it conforms to the respective `web` tool.

- The need for a tool that enables code-reordering, which has been one of the most highlighted features of all `web` derivatives, is eliminated by modern implementation languages. Functional and logic programming languages provide this feature inherently without an additional "weaver".

- Although Knuth used his `web` to document the implementation of TEX (which can be considered to be quite a large and complex piece of software), none of the currently known `web` derivatives supports modularization or the hierarchical organization of groups of modules. These large-scale structuring concepts, having proved suitable for practical software development, should be supported by a documentation tool.

Some of our ideas were motivated by the GRASP Literate Programming System, which was developed at the University of Glasgow [GRA92] on the basis of the functional programming language HASKELL. Although GRASP also lacks a concept for what we call "large-scale, structured documentation" support, the removal of emphasis on code-reordering and the simple syntactic embedding of Haskell code into the LATEX typesetting language conform to our ideas about appropriate source-code documentation.

GRASP – as well as all other `web` offspring – focusses on minimizing the problem of code/documentation consistency by reducing the spatial distance between source code and explanatory text. Conventionally, source code and documentation are managed in at least two separate documents (e.g. program listing

and informal documentation handbook). By contrast, the interweaving approach has been adopted in tools for formal requirements specification where the explanation of formal notation plays a central role. The Z specification language, in connection with the fuzz-package for typesetting and type-checking developed by M. Spivey at Oxford University [Spi92b, Spi92a], serves as an appropriate example. As in GRASP, small pieces of formal description are embedded into the informal context provided by a document containing plain text.

2.2 Objectives of the DOSFOP System

In this section, we want to present our essential goals in implementing the DOSFOP documentation system. Most of our objectives derive from experiences with existing documentation tools, as described above, and are motivated by the fact that acceptance of the documentation system by the program developer has to be our main focus of interest.

Exploit inherent documentation elements as much as possible

The notion *inherent documentation* refers to all the kinds of design decisions, naming conventions, structuring aspects, etc. that come up during "conventional" software development and are useful for documentation purposes. For example, the module hierarchy built up while developing large programs can be used as a basis for the structure of both a printable and an online hypertext documentation. It can be seen as a generated skeleton of an individually extendable documentation.

The definition and application of program identifiers is another main concern the programmer has to deal with conscientiously. We should use the programmer's work as a basis for the administration of various cross references automatically generated by the documentation tool. As we see it, support for *browsing* has to be one of the central elements of a really useful system, and it can be achieved by reusing the design information needed to produce a running (and perhaps completely undocumented) implementation anyway. Especially in documentation represented in hypertext, tracing of references to definition or application positions of identifiers is an example of useful exploitation of inherent documentation information.

Provide a convenient documentation environment

Nowadays, integrated programming environments are becoming more and more powerful and much easier to use (e.g. by the application of high-resolution graphics, window systems and intuitive point-and-click interfaces). A documentation system has to cope with the existing program-

ming environments and, moreover, must motivate software developers to produce documentation as a side effect of software compilation.

First, the system has to provide support while the documentation is under construction. Because implementation is normally done using editors with additional features tailored with respect to a specific implementation language, we have to embed our system into this environment and add some supplementary documentation features to the editor.

Second, the use of familiar mechanisms for explaining a program's source code simplifies the arduous task of writing explanatory text. The commentary conventions of the programming language chosen for the project serve as an appropriate notational starting point. The great advantage of commentaries is the resulting availability of spatial interconnection with the documented code, which lessens the problem of consistency between code and documentation.

Finally, documentation already produced has to be easy to "debug", both with respect to the content and layout of the written text as well as to the structure and extent of the documentation as a whole. This can be achieved by means of existing previewers for mark-up text-formatting languages like LaTeX and HTML and by rapid generation of the documentation for the purpose of immediate feedback.

Keep the documentation configurable

For the adoption of the structure and extent of a (partly) generated documentation, a special user interface is needed that offers all choices and possible combinations of documentation elements (e.g. production of indices, ordering of modules, hiding special implementations or subsystems, etc.). This is what we call *configurable documentation.*

Do not sacrifice old code

When introducing a new documentation tool into an established programming environment, one has to enable a smooth transition from the conventional to the innovative documentation approach. Consequently, a fundamental requirement is that undocumented "old" code can be incorporated using the features of the new documentation system.

As an extension of this principle, the introduction of the documentation tool should not change the behaviour of the other components of the programming environment.

Allow the documentation of large, structured implementations

Existing documentation environments are generally not designed for handling large-scale implementations that are hierarchically structured into modules and subsystems. These are the type of implementations we are most interested in, and approved mechanisms for *programming in the*

large are considered to form the basis of each generated documentation.

Provide different kinds of documentation presentation

Source-code documentation can be helpful to persons coming from a variety of backgrounds and each with a different interest in the project's implementation. As a consequence, documentation has to do justice to many requirements. It would be illusory to try to produce different kinds of documentation products that are each tailored to specific needs. This last statement is confirmed by the experiences of the GRASP team ([GRA92] p. 48 "Death to ribbons..."). Our approach is to enable the programmer to present his or her documentation in different ways, at no extra effort.

We believe that *online documentation in hypertext format* is well suited for browsing: persons interested only in particular areas of the implementation can comfortably zero in on what they want, while inexperienced novices can follow the predefined standard browsing paths. Readers who have difficulties with online hypertext presented on a monitor can refer to a high-quality printed documentation. Due to the sequential nature of this media, one has to use a table of contents and indices for manual browsing (turning the pages).

Motivate the user with convincing products

Minor obstacles (which cannot be completely prevented if useful documentation is to be produced) can be be overcome if the resulting documentation is found to be practicable by the customers, the person in charge of the project or, last but not least, the programmer him- or herself. Only appreciation of the product will motivate the programmer to put more effort into the informal explanation of source code at software development time. When designing the features of a documentation system, we always have to keep in mind that we need an impressive end product.

We will return to this catalogue of objectives after the description of the DosfOp system and discuss to what extent the goals have been achieved.

3 Description of the DosfOp System

This section contains a description of the DosfOp system. Following the introduction of some terminology, in the second section we will describe the com-

ponents of the documentation system. In the following section, the integration of documentation into the source code — the core of literate programming — is discussed. The final part contains a description of special features provided by the DosfOp system for customizing the output.

3.1 Terminology

In describing DosfOp, we do not use features that are specific to Opal, but we use some terminology from the Opal environment, which we explain here briefly.

The DosfOp system uses comments from the programming language to incorporate documentation. The Opal syntax for comments that span several lines is `/* ...*/`. The notation for comments that end at the end of the line is `-- ...`.

A *structure* in Opal corresponds to "modules" or "packages" in other languages. A structure consists of four *parts*: the *signature*, the *implementation*, the *external properties* and the *internal properties*; each of these parts corresponds to a separate file. The property parts may be omitted. The signature and the external properties together constitute the *interface* of an Opal structure.

A *subsystem* is a collection of structures and possibly other subsystems. The concept of a subsystem corresponds roughly to the notion of "library" in other languages.

An Opal program has a *top function*, which is a monadic command that controls the execution. The structure where the top function is located is called the *top structure* of the project. The place where the top structure is located within the DosfOp data base is called *toplevel*.

3.2 Components of the DosfOp System

The general idea of literate programming is straightforward: feed the literate program into the documentation generator and out comes the documentation. The DosfOp system, however, has projects as input, allows customization and produces several output formats. Hence, with DosfOp, things are not quite so simple.

On the input side, DosfOp not only needs the source code, because DosfOp documents not only single files but also larger Opal projects and the modularization should be reflected in the documentation as it is provided in the program. DosfOp therefore additionally needs a *project data base* in which

the structure of the OPAL project is recorded. Moreover, DOSFOP does not produce a uniform documentation format – the user has many possibilities to customize the result via *global options*.

On the output side, DOSFOP does not directly produce documentation, rather it produces an intermediate output file in the TEXINFO language. This intermediate output is translated in turn into a final representation. The advantage of this approach is that we do not need to write the necessary translators, rather can use existing tools. Currently, DVI files (for printed output), INFO files (for the GNU info help system) and HTML files (for web browsers) are supported.

Figure 1 is a graphical representation of the generation of a documentation with DOSFOP.

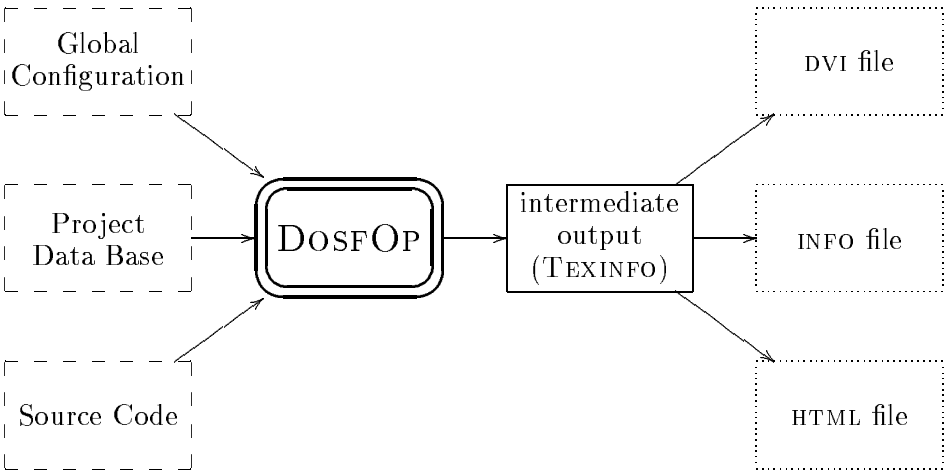

Figure 1: Producing documentation

3.3 Example of Generated Documentation

Before we proceed with the description, we want to present a short example. Figure 2 on the following page shows a piece of literate source code from the implementation of a small SKI-combinator interpreter, Figure 3 shows the corresponding printed output, while Figure 4 was generated by a WWW browser. The HTML hypertext is divided into one node for every section or subsection.

```
/* %This structure provides an @sc{Opal} interface to parsers generated
with yacc. There are three parsers available for parsing
@itemize @bullet{}
@item
standard input
@item
or (the contents of) some file
@item
or a denotation.
@end itemize
*/

SIGNATURE LexYacc[data]

-- %$Parameter$
SORT data
/* %
@code{data} is usually an abstract syntax type. It must be the same type
that is constructed by the yacc actions. Since the interface is handcoded,
no type checking is possible. You may encounter weird error messages if you
supply a parameter type which is different from that used in the yacc parser.
*/

-- %$Imports$

IMPORT Com[data] ONLY com
       File ONLY file

-- %$Parsing$
/* %
All of the functions deliver a "fail" answer if a parse error
occurs. This answer contains the message generated by the parser.
*/

FUN parse: com[data]


FUN parse: file -> com[data]


FUN parse: denotation -> com[data]
```

Text enclosed in `/* ...*/` is an OPAL comment. A leading per cent sign (`%`) marks a text as DOSFOP documentation. Words starting with an "at" sign (`@`) are TEXINFO commands.

Figure 2: Source code

4.2 LexYacc

This is the structure which performs the connection between the generated parser and OPAL.

Three different functions - all called `parse` - provide access to this parser. You can parse either

- standard input
- or (the contents of) some file
- or a denotation.

4.2.1 Signature of LexYacc

```
SIGNATURE LexYacc[data]
```

4.2.1.1 Parameter

```
SORT data
```

`data` is usually an abstract syntax type. It must be the same type that is constructed by the yacc actions. Since the interface is handcoded, no type checking is possible. You may encounter weird error messages if you supply a parameter type which is different from that used in the yacc parser.

4.2.1.2 Imports

```
IMPORT Com[data] ONLY com
       File ONLY file
```

4.2.1.3 Parsing

All of the functions deliver a "fail" answer if a parse error occurs. This answer contains the message generated by the parser.

```
FUN parse: com[data]

FUN parse: file -> com[data]

FUN parse: denotation -> com[data]
```

Figure 3: Printed output

Go to the first, previous, next, last section, table of contents.

LexYacc

This is the structure which performs the connection between the generated parser and OPAL.

Three different functions - all called `parse` - provide access to this parser. You can parse either

- standard input
- or (the contents of) some file
- or a denotation.

- LexYacc Sign
- LexYacc Impl

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Signature of LexYacc

- LexYacc Impl: IMPLEMENTATION LexYacc
- Parameter (Part of LexYacc Sign)
- Imports (Part of LexYacc Sign)
- Parsing (Part of LexYacc Sign)

```
SIGNATURE LexYacc[data]
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Parameter

```
SORT data
```

`data` is usually an abstract syntax type. It must be the same type that is constructed by the yacc actions. Since the interface is handcoded, no type checking is possible. You may encounter weird error messages if you supply a parameter type which is different from that used in the yacc parser.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Parsing

All of the functions deliver a "fail" answer if a parse error occurs. This answer contains the message generated by the parser.

```
FUN parse: com[data]

FUN parse: file -> com[data]

FUN parse: denotation -> com[data]
```

Go to the first, previous, next, last section, table of contents.

The corresponding hypertext is divided into five different nodes, four of which are shown here.

Figure 4: Hypertext

3.4 Using DosfOp

To use DosfOp, the input must be set up as depicted in Figure 1, that is, the user must provide the Global Configuration, create the Project Database and supply the source code. The last task is the easiest one because DosfOp requires no changes to the source code. Of course, the source code is the place where the documentation should be added (see Section 3.5), but the system does not require the code to be adapted.

Initializing DosfOp requires setting up the Global Configuration and the Project Data Base. The Global Configuration need usually not be changed again, whereas the Project Data Base must be kept up to date by adding or deleting structures.

3.4.1 The DosfOp Main Window

The DosfOp Main Window (see Figure 5) very much resembles the picture in Figure 1. It is the "head office" for invoking tools to edit the Global Configuration and the Project Data Base, for invoking the DosfOp translator itself, for generating the DVI file, INFO file or HTML file, and for calling up a previewer or browser to view the documentation on the screen.

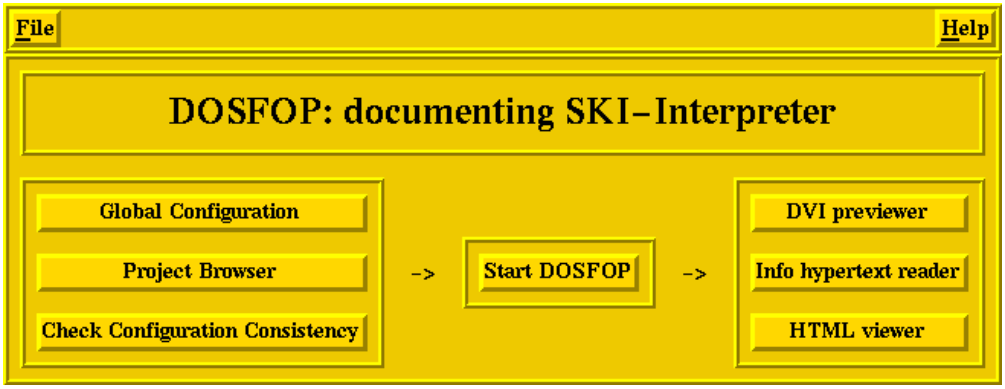

Figure 5: DosfOp Main Window

3.4.2 Global Configuration

In the Global Option Configurator (see Figure 6), the root location of the project, the intermediate output file and the name of the top structure have to be provided. For the title page of the documentation, one can additionally add the project name, the authors' names and the date of creation. There are many other options available: for a description, see Section 3.6.1.

DosfOp provides sensible defaults but cannot guess the correct name of the top structure, which must be entered by the user. The top structure[3] and all the structures in the root directory of the project – and possibly also user-defined subsystems that are directly or indirectly imported by it – are included in the documentation (with the exception of files from the standard library).

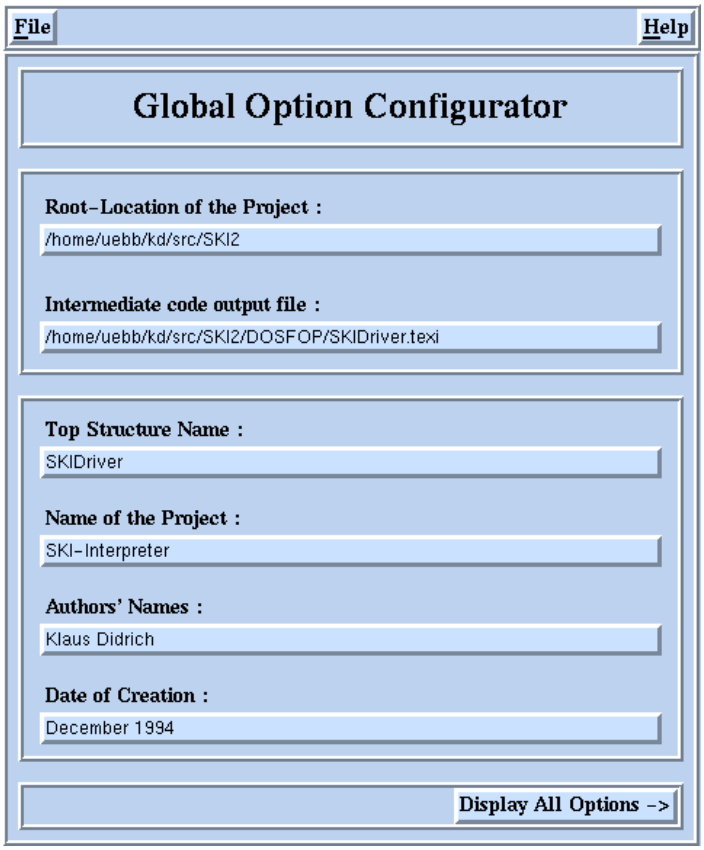

Figure 6: Global Option Configurator

3.4.3 Project Browser

The Project Browser manages the DosfOp data base. This data base contains information on all structures and subsystems that are used within the documented Opal project (except the structures from Bibliotheca Opalica).

Most important is the information as to where structures and subsystems are located, that is, in which subsystem they are to be found, and the path name to access the source code. An additional feature is the configuration of subsystems and structures, which may override the global configuration.

The Project Browser is a kind of tree editor for an Opal project, where the user can insert, delete and rename structures and subsystems. Subsystems

[3] It is also possible to process projects without a top structure, such as libraries.

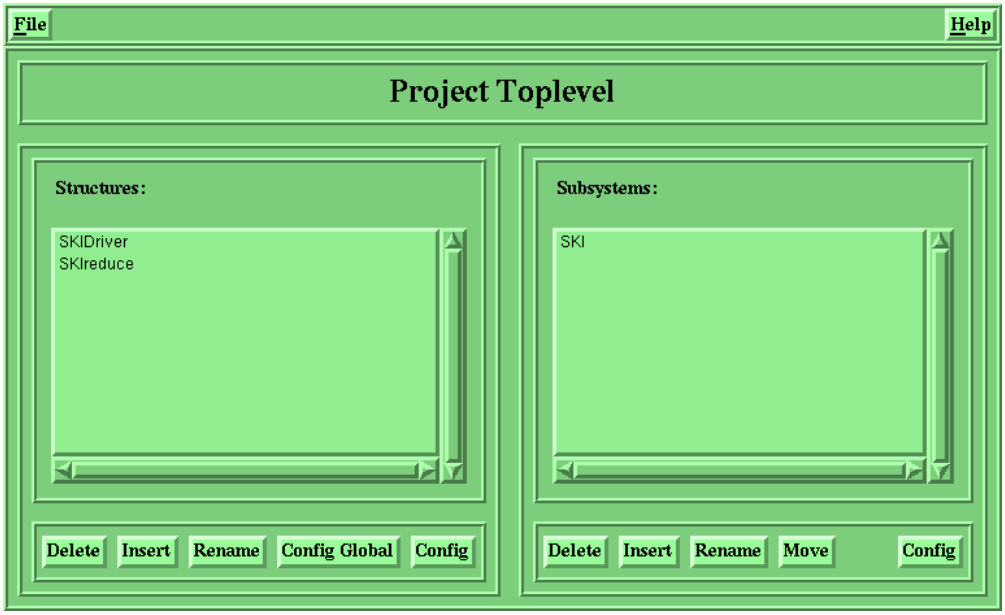

Figure 7: Project Browser

can also be ordered by the user, whereas the order of structures is determined by DosfOp according to customizable options. The names of structures are determined by Opal, but the names of subsystems are managed by DosfOp and can be chosen freely.

3.4.4 Generating Documentation

Once everything is set up, the user can initiate the generation of the documentation. DosfOp does not immediately produce all possible output formats, rather asks the user which formats should actually be generated. The following output formats are currently available:

printable documentation / dvi format If printable documentation is required, DosfOp generates a DVI file, which can be previewed and printed.

hypertext / info format Hypertext in INFO format, which can be used, for example, in the GNU emacs editor.

hypertext / html format A family of files in HTML format, which can be viewed with any WWW browser.

3.5 Writing Documentation

This section contains information about writing documentation using the DosfOp system. We first describe the language the documentation is written in,

then show how project and subsystem surveys are included and, finally, explain how documentation is integrated into the OPAL source code.

3.5.1 A Word About Texinfo

TEXINFO [CS93] is the language in which the documentation is written by the user. TEXINFO and its associated tools are freely redistributable software, which was developed by the Free Software Foundation and is used for the documentation of GNU utilities and libraries. The TEXINFO language is designed to generate printed and hypertext documentation from a single document and contains support for index generation.

From the user's point of view, TEXINFO has the look and feel of a remote dialect of TEX, without the possibility of declaring one's own commands, and without the multitude of styles. The documentation examples in Figure 2 and Figure 9(a) show text written in the TEXINFO language. For a documentation of TEXINFO see [CS93].

3.5.2 Project, Subsystem and Structure Surveys

Usually, the documentation is included in OPAL source code; after all, this is the idea of literate programming. There is, however, no place in OPAL code that corresponds to a whole project or a whole subsystem. You can insert "surveys" of (or introductions to) the project and subsystems into the documentation by configuring them.

The *Project Survey* is managed by the extended Global Option Configurator. A *Subsystem Survey* is inserted by a Subsystem Configurator.

Structures may also have a survey. As every OPAL structure must have a signature part, we decided the first documentation in the signature part of a structure should be the *Structure Survey* if the documentation is located before the keyword "`SIGNATURE`".

3.5.3 Source-Code Documentation

All other documentation is contained in the OPAL source code in the form of special comments. Every comment that starts with a per cent sign (`%`) is treated as documentation. All other comments are treated as source code. Note that both types of comments (see 3.1) are treated alike.

We found it useful to distinguish between various kinds of documentation. In DOSFOP, there are four[4] different kinds of documentation. We present them in turn, each with an example in concrete DOSFOP documentation syntax.

[4]There is actually also a fifth kind of documentation available, which is specific to OPAL. See Section 3.6.3.

Ordinary Documentation is any documentation that does not belong to the following categories. Ordinary documentation may contain arbitrary (TEXINFO) text.

```
-- %This function computes the length of a sequence.
```

Tagged Documentation Tagged documentation is ordinary documentation with an additional tag. This documentation does not appear in the generated documentation unless explicitly specified in the global configuration. This feature can be used to generate documentation for different audiences.

```
/* %{optimize} This function needs further attention
   if used with bigger arguments.
*/
```

Documentation Sectioning In addition to the structuring provided by DOSFOP, it is useful to insert additional headings in the source code. We cannot use TEXINFO commands for this purpose because they define the absolute sectioning level.

The number of dollar signs (`$`) at the beginning indicates the *relative* sectioning level of the heading. The current implementation of DOSFOP allows up to four additional levels within a source-code file.

```
-- %$$Accessing elements of a sequence$
```

Hidden Documentation Every system has its comments, and so does DOSFOP. Unlike OPAL comments, hidden documentation does not appear at all in the generated documentation.

```
-- %- Check below carefully for typos!
```

3.6 Special Features

The DOSFOP system provides a few special features that enable the user to write documentation in accordance with personal preferences. They concern the following areas:

- Customizing the documentation
- Structuring the documentation
- OPAL speciality: property references
- TEXINFO setup

3.6.1 Customizing the Documentation

DosfOp provides several possibilities for changing the appearance of the generated documentation. The documentation can be configured differently on three levels: global, subsystem-local and structure-local.

Configuring on the global level is done via the Global Option Configurator. These customizations are valid by default for the whole documentation. Table 1 explains the switches that are available.

`Tagging`	Enable tagged comments.
`Functionality Index`	Include an index of all functions together with their functionalities.
`Application Index`	Include an index of all applied functions.
`Concept Index`	Include a concept index. The user must insert `@cindex` ***`entry`*** lines manually.
`Structure Index`	Include an index of all structures.
`Subsystem Inclusion`	Include only structures on the toplevel, if switched off.
`Library Inclusion`	Include interfaces (signature and possibly external properties parts) of referenced structures from Bibliotheca Opalica.
`Property Inclusion`	Also include property parts.
`Include only Interfaces`	Include only signature parts of structures (also external property parts, if property inclusion is switched on).
`Hierarchy Visualization`	Include a graphical representation of the hierarchy of subsystems and/or structures
`Sort Structures`	Sort structures within subsystems top-down, bottom-up, ...
`Import Referencing`	Include references to imported structures.
`Used Function Tables`	Include tables that show which functions are applied in function definitions.
`Basic Language`	Choose the language of the generated text.
`Start New Page`	Start new page for every structure part in the printed documentation.
`Drop Empty Lines`	Remove empty lines at the end of source code.

Table 1: Brief description of global switches

Customizing locally for a subsystem is done via a Subsystem Configurator. This configuration overrides the global configuration and is valid for structures and subsystems contained in the subsystem, unless they are configured differently themselves. The possible options are a subset of those presented in Table 1.

Finally, it is also possible to *configure a single structure* with a Structure Configurator. This configuration in turn overrides both the global and the subsystem local configurations. The possible options are again a subset of those listed in Table 1.

3.6.2 Structuring the Documentation

While the structure of the OPAL project is a good starting point, DOSFOP does not require that the structure of the documentation mirror the physical structure of the project in the file system. The only requirement is that every structure referenced in the project must either be a structure from BIBLIOTHECA OPALICA or be located somewhere in the DOSFOP data base.

This opens the possibility of subdividing a subsystem further if such a structure is more suitable for representation purposes. You can also completely reorganize the physical structure if that leads to better documentation. Depending on the readers targeted, this reorganization may or may not be a good idea. A different structure might be the most appropriate way to document the interfaces of a library. If the documentation is not for the user but for the administrator who maintains and debugs the code, the structure of the documentation should resemble the physical structure.

3.6.3 An OPAL Speciality: Property References

Property references are an experimental feature. They allow inclusion of laws from the property parts in the documentation simply by referencing their names.

Syntactically, property references consist of a list of names of laws from the external or internal property part of an OPAL structure. In the documentation, these references are replaced by the law from the corresponding property part.

Figure 8(a) shows the source code of a concatenation function with a reference to a law from the property part. Figure 8(c) shows the generated printed output. (The formula is derived from the abstract syntax, which does not reveal information whether an application was written in prefix, infix or postfix notation. Therefore, the notation of the law differs from the source code shown in Figure 8(b) .)

```
FUN ++ : seq ** seq -> seq
-- %[++_assoc]
```

(a) source code in signature part

```
LAW ++_assoc == ALL a b c. a ++ (b++c) === (a++b) ++ c
```

(b) source code in external properties part

```
FUN ++ : seq ** seq -> seq
```

$$\forall a, b, c. + +(a, + + (b, c)) \equiv + +(+ + (a, b), c)$$

(c) printed output

Figure 8: Property references

3.6.4 TEXINFO Environment

There may be situations in which you do not want to use the same text in the printed and the hypertext versions of the documentation. In the printed documentation, one can use the capabilities of TEX, which is particularly suited to mathematical formulae. In the hypertext, however, one has to provide a textual representation of formulae.

As an example, see the documentation of the postcondition of a numerical integration function. The source code in Figure 9(a) shows how to write different documentation for printed and HTML output. Figures 9(b) and 9(c) display the corresponding results for the two different outputs.

DOSFOP extends TEXINFO slightly by providing a *macro definition* facility. Macros are set globally for a document and the Macro Editor is thus selected via the Global Option Configurator. Within the Macro Editor you can define simple macros for use in your documentation.

DOSFOP sets TEXINFO up such that you can directly use the ISO Latin 1 character set (ISO-8859-1), which is important if the documentation language

```
Compute an approximation of
@ifset html
the integral of @code{f} between @code{a} and @code{b}.
@end ifset
@tex
$\int_a^b f(x) dx$.
@end tex
The result is a list of better and better approximations.
The @i{n}th element of the list is the approximation with
@ifset html
@i{2^n}
@end ifset
@tex
$2^n$
@end tex
intervals.
```

(a) source code

Compute an approximation of $\int_a^b f(x)dx$. The result is a list of better and better approximations. The nth element of the list is the approximation with 2^n intervals.

(b) printed output

Compute an approximation of the integral of `f` between `a` and `b`. The result is a list of better and better approximations. The *n*th element of the list is the approximation with *2^n* intervals.

(c) hypertext output

Figure 9: Usage of conditional documentation

is not English. The only other language supported is german, but others can be easily added.

4 Experiences

Since its creation, the documentation system has been applied to several small and medium-sized programs:

- The prototype of the "*Studienarbeit*" implementation of DosfOp was used for the documentation of Opal programs in a joint project with Daimler-Benz AG Forschung Systemtechnik.
- The extended DosfOp implementation described in the master's thesis [Kle95] was applied
  - in the documentation of the ESZ[5] project,
  - in the documentation of OpalWin[6][FGPS96], and
  - of course, in the documentation of DosfOp itself.

We first discuss to what extent the requirements listed in the catalogue in Section 2.2 are met.

The DosfOp system relies on some tools developed by other people who never dreamed that their products would be used to document Opal projects. We discuss some of the difficulties we encountered with these tools below.

The system has some flaws that could be improved on, if only we could devote more time to the further development of DosfOp.

4.1 Critical Discussion of the Objectives

In this section we review the objectives singled out in Section 2.2, discuss whether these objectives are met by our implementation and, if so, whether the desired effect has been achieved as a result.

Exploit inherent documentation elements as much as possible We mentioned two applications for this principle. First, the module hierarchy as the basis for the documentation structure and, second, the support for browsing by automatically generated indices.

We found that printed output and hypertext output require separate treatment. The documentation structure is already handled well by DosfOp and Texinfo;

[5] ESZ is a type-checker for the Z formal specification language.

[6] OpalWin is an Opal library for a graphical user interface based on concurrent functional programming.

however, the HTML output suffers from the fact that the tool used does not fully preserve the hypertext structure generated by DosfOp.

The generated indices are also used differently. Functional languages tend to support a style in which many small items are written. The indices are easy to use in hypertext because all entries are links to corresponding declaration. In printed output, the references to pages are too unspecific, to be really useful. Surveys and overviews, on the other hand, are more effective on paper.

Provide a convenient documentation environment The addition of new documentation by means of specially marked comments turned out to be the most natural way to incorporate documentation.

The aim of easy debugging is only partially met. The documentation system is only superficially integrated into the Opal compilation system and is therefore slower than desirable. The documentation must be processed as a whole, while the compilation is done in small steps; so the time invested in generating the documentation is considerably longer than he time needed for compiling the program. While the time is acceptable for printed output, the more volatile hypertext should ideally be produced more quickly.

Keep the documentation configurable The configurability of DosfOp has contributed much to its acceptance. We already mentioned that ordering of structures within a subsystem very much depends on the author's preferences. Other issues like choice of language, used-function tables and the optional inclusion of structure parts also play a role in the adaptation of the output to the user's needs. We strongly believe that hard-wiring of these options would result in a documentation system only its author(s) would use.

Do not sacrifice old code DosfOp allows a smooth transition from undocumented structures to a documented project and does not force the programmer to "surrender" unconditionally to the documentation system. The programmer can decide at any point whether the program has reached the stage where it is worth documenting it properly. The decision to document a software system does not commit one to the use of DosfOp. One can always stop documenting and still compile the code without difficulty. This was important in the early days of DosfOp, when we did not know whether the documentation system would finally work or not. It is still important for users who are suspicious about documentation systems.

Allow the documentation of large, structured implementations DosfOp has been successfully applied to a number of software projects, the largest being the DosfOp system itself and Bibliotheca Opalica, Opal's standard library. With more than 150 structures, organized in several subsystems, the

generation of documentation for the standard library has been an endurance test for the documentation system.

Provide different kinds of documentation presentation The documentation system provides three different kinds of output. The info hypertext is only rarely used; the HTML hypertext provides almost the same functionality with a better-known user interface. Hypertext documentation is mostly used for libraries that are accessed frequently.

Printed documentation is still the most important kind of output, perhaps because the art of producing hypertext that is easy to read is not as well developed as the art of producing printed text. This will improve in the future, we hope, and because it is easy to integrate different translators into the DosfOp system, we will profit immediately from new developments in this area.

Motivate the user with convincing products The printed documentation produced by DosfOp follows the style developed by FSF for printed documentation. This style has proved practical for our purposes, and the documentation has impressed many other people. The INFO hypertext, however, suffers from the fact that the tool is text oriented. We are also not fully satisfied with the generated HTML hypertext, because the translator does not preserve the node structure of the intermediate Texinfo code. Still, the hypertext provides a tool for easy browsing and access to declarations of functions and sorts.

4.2 Experiences From An Implementor's Point of View

4.2.1 Application of Third-Party Tools

The DosfOp system relies on a number of programs from third parties, most notably Tcl/Tk and the software from the Free Software Foundation (FSF) for processing Texinfo. Most of these programs were not developed with the documentation of Opal programs in mind, and we consequently encountered some difficulties.

Tcl/Tk The implementation of DosfOp consists of a Tcl/Tk part and an Opal part. Some problems in installing the system result from the Tcl/Tk part, which requires a secure X server[7]. Thus, some caution is required in calling up the window manager.

The requirement of a secure X server seems reasonable enough, but we encountered quite a few non-secure servers. Because most people do not know how to set up their server in a secure way, this is perhaps a reason why the system is not used.

[7] They are caused by the Tcl command **send**.

The Texinfo Language The choice of TEXINFO as the documentation language is controversial. The language itself is not difficult, and one needs only a small part of it to write documentation. But TEXINFO is not a very popular language, users preferring languages such as LaTeX or, in the age of the World Wide Web and its browsers, HTML. There are now some LaTeX-to-HTML translators (e.g. `LaTeX2HTML`, `Hyperlatex`), while other approaches try to extend LaTeX by URL hyperlinks in special previewers (e.g. `HyperTeX`). However, most of these tools try to translate a paper document automatically into a hypertext, instead of integrating both views in one language, like TEXINFO does.

One other possibility would have been to invent a documentation language of our own, but we wanted to concentrate on the documentation system and not invest too much effort in the development and maintenance of a new language. One could also introduce syntactic sugar and make the language look like LaTeX. But experience with the GRASP system shows that users tend to forget that they are not using LaTeX, rather only a restricted subset.

In spite of all the inexpediences mentioned above, TEXINFO is a language that is well suited as a mark-up language for formatting text and as an intermediate language for the generation of printed documentation and documentation in hypertext form. TEXINFO has the best built-in support for handling multiple indices. Some of the points mentioned above cannot be amended easily, because TEXINFO must be translatable to two[8] different languages and is therefore often restricted to the least common denominator.

The Texinfo Tools The tools that translate TEXINFO into the respective target languages are also not perfect. None of these tools really does check that the input conforms to the description in [CS93]. Most tools silently correct minor errors, especially in the parts that are not important for the target format. As a consequence, the user gets different error messages from the different tools and has difficulty matching the error to the location in the documentation.

Some tools (`texindex` is an example) generate output that causes formatting problems in the context of DOSFOP. These problems are treated by auxiliary programs.

4.2.2 DOSFOP Implementation Aspects

Error-Handling After installation there are no problems — provided the user makes no mistakes. DOSFOP's error-handling is currently the weak point: users would like to have a tool with elaborated error recovery. The OPAL compiler, by contrast, has much more sophisticated error-handling and the documentation system is inferior in this respect, which irritates users.

[8] The HTML format was added later, and the translator from TEXINFO to HTML is not supported by the Free Software Foundation (FSF).

Partitioning of the System The partitioning of the system turned out to be an obstacle to system enhancement. While it was a straightforward affair to enhance the documentation system itself, the adaptation of the interface part was difficult. The fact that the user interface and the DosfOp system are implemented in two different languages contributed to the problem. A potential option is to reimplement the user interface in Opal. This is now possible with the recently developed window library for Opal [FGPS96].

An Opal user interface would also be a solution to a further criticism. The configuration chosen by the user is communicated via an ASCII interface language, which is then scanned and parsed again by the proper documentation system written in Opal. If the interface were also written in Opal, the time-consuming three-stage process would not be necessary.

Separate Tool DosfOp was developed as a separate tool to be added to the Opal environment. This means that the tool has to be called up separately, and — even worse — that addition (or deletion) of a structure of the software project must be done twice: once for the Opal compilation system and once for DosfOp. We thought this would be an advantage because the documentation could then have a different structure to the actual system.

Experience showed, however, that this feature is only rarely used. And the user is annoyed by having to register each change of the modularization with both the Opal compilation system and the documentation system. Thus, we have come to the conclusion that we need to integrate the documentation system into the overall compilation system as well, so that the button that starts the compilation of the program also starts the generation of the documentation.

Efficiency We mentioned already that the documentation must always be processed as a whole, and that generation is therefore slow compared to compilation. The first possible remedy is to generate a piece of documentation for every module; to produce the documentation, one simply concatenates these pieces. However, even if this were possible, there is the problem that the user might change some switches that affect the pieces already generated. The webbing of the pointers to and from other nodes of the hypertext document is especially problematic.

Creation and management of modularized text is a fairly new topic (see e.g. [GB93]) and does not fall within the scope of the research activities carried out in our department.

5 Conclusion and Future Work

The difficulties mentioned in the previous section do not render the system unusable, but only a user who is convinced of the advantages of good documentation and is willing to take time to get acquainted with the system will use it.

If these conditions are given, DosfOp provides powerful support in reducing the administrative overhead of documenting software. Of course, the text has to be written manually. In an ideal programming environment, this would be the only effort required, and we hope that the integration of DosfOp into the Opal compilation system will eventually yield such an environment.

What is the result of the introduction of the documentation system? Code is better documented because the user feels that the result is worth the little extra effort of formulating a more explicit commentary in the form of a documentation. We hope that the further development of DosfOp will produce a system that will be able to generate an online documentation as a side effect of compiling, while minimizing the delay in the compilation process itself.

Future work will concentrate on extensions and the integration of DosfOp into the Opal environment.

Error-handling in DosfOp needs to be improved by a robust error-handling scheme that reports errors together with a precise description and then proceeds with the generation of text.

The integration of the Opal compilation system and the documentation system cannot be done in a straightforward way (as mentioned above) because the generation of Texinfo code depends on settings that are only known at the time the overall documentation is generated. But if one accepts some inconsistencies in the course of program development, this goal might be achieved. One would have to clean up the inconsistencies once in a while of course. The gain would be the possibility of generating object code and documentation together and always having both automatically.

The Texinfo documentation format has developed since the completion of DosfOp. The most recent version [CS96] contains built-in support for the enhancements described in Section 3.6.4. Some commands have been introduced which facilitate the modular generation of the intermediate Texinfo code.

Finally, let us mention that the abstract syntax of Opal 2 [DEG+96], which is currently under development, has been designed to support DosfOp (or DosfOp 2) by including comments and documentation in the concrete *and* the abstract syntax, so that the new Opal 2 system will, from the very beginning, have an integrated documentation system.

Acknowledgement The work described in this report profited from discussions with other members of the Opal Group and from participants of the projects in which the first prototypes were used. We are grateful to Niamh Warde for her valuable help in proof-reading this report.

References

[AO90] Adrian Avenarius and Siegfried Oppermann. `FWEB`: A literate programming system for Fortran 8X. *ACM SIGPLAN Notices*, 25(1):52–58, January 1990.

[CS93] Robert J. Chassell and Richard M. Stallman. Texinfo — The GNU Documentation Format. In hypertext form available at URL `http://www.cl.cam.ac.uk/texinfodoc/texi_toc.html`, March 1993. Edition 2.18, for Texinfo Version Two.

[CS96] Robert J. Chassell and Richard M. Stallman. Texinfo — The GNU Documentation Format, October 1996. Edition 2.23, for Texinfo Version Three.

[DEG+96] Klaus Didrich, Jürgen Exner, Carola Gerke, Wolfgang Grieskamp, Christian Maeder, Peter Pepper, and Mario Südholt. The OPAL 2 Language — Alpha Version. Technical report 96-3, TU Berlin, 1996. To appear.

[DFG+94] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In J. Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 228–244. Springer, 1994.

[FGPS96] Th. Frauenstein, W. Grieskamp, P. Pepper, and M. Südholt. Concurrent Functional Programming of Graphical User Interfaces. Technical Report 95-19, TU Berlin, 1996.

[GB93] Michael J. Groves and David F. Brailsford. Separate compilation of structured documents. *Electronic Publishing — Origination, Dissemination, and Design (Journal)*, 6(4):315–326, December 1993.

[GRA92] The GRASP Team. Glasgow Literate Programming User's Guide, September 1992. Contact: Will Partain.

[KL93] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison-Wesley, Reading, MA, USA, 1993.

[Kle95] Torsten Klein. DOSFOP – Ein benutzerfreundliches Dokumentationssystem. Diplomarbeit, TU Berlin, April 1995.

[Knu83] Donald E. Knuth. The WEB system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[Knu84] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Knu91] Donald E. Knuth. *The TEXBook.* Addison Wesley Publishing Company, Stanford University, 1991.

[Opa95] The Opal Group. The OPAL Project. Available at URL `http://www.cs.tu-berlin.de/~opal/`, 1995.

[Ram89] Norman Ramsey. Weaving a language-independent WEB. *Communications of the Association for Computing Machinery*, 32(9):1051–1055, September 1989.

[Ram92] Norman Ramsey. Literate-programming tools need not be complex. Report at `ftp.cs.princeton.edu` in `/reports/1991/351.ps.Z`. Software at `ftp.cs.princeton.edu` in `/pub/noweb.shar.Z` and at `bellcore.com` in `/pub/norman/noweb.shar.Z`. CS-TR-351-91, Department of Computer Science, Princeton University, August 1992. Submitted to *IEEE Software*.

[Spi92a] J. M. Spivey. *The fuzz Manual*. Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.

[Spi92b] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Van90] Christopher J. Van Wyk. Literate programming—an assessment. *Communications of the Association for Computing Machinery*, 33(3):361, 365, March 1990.