From logic to manuals

by Harold Thimbleby and Peter Ladkin

A simple language is demonstrated that combines specifications and manuals. This shows first that a user manual can be automatically reconstructed from a logic specification that is effectively identical to the original logic (up to ambiguities in natural language); and secondly, that such an automated process can help detect errors. The process is fast and suitable for use in participatory design.

1 Introduction

To use complex systems safely and efficiently requires appropriate training and documentation, based in turn on an understanding of the systems themselves. Systems are (or should be) based on a specification of what they are to do. We take a requirements or design specification henceforth to be a (logical) predicate in a precise language that a system is required to satisfy. This specification puts conditions on certain state variables of the system. A system satisfies the specification just in case it does not exhibit behaviour that is proscribed by the specification, and it fails to satisfy a specification just in case it may exhibit behaviour that is proscribed by the specification. How do we tell the users how to use the system? How do we ensure that our instructions are logically compatible with the specification?

It is widely recognised that requirements change and systems evolve, among other reasons because of design needs and in response to user feedback [1, 2]. In systems where users form an essential part, it is common practice to involve the users in the design process. As users increasingly understand a system, they may help change its design for the better. Logically accurate user documentation is desirable in itself, and a reliable, fast method for generating it could improve the design-trial-redesign loop. We describe a simple prototype tool that accomplishes this, implemented by the first author.

Aircraft control and flight management is a human-in-the-loop system. We use it as an example, partly because some actual user manual pages have been reverse-engineered by the second author [3] and now exist in a form on which we can demonstrate the forward-generation of new manual pages; and partly because it is pilot folklore that flight crew operating manuals (FCOMs) are in constant need of improvement [4]. However, the technique we illustrate is applicable to the derivation of accurate documentation in general, and the tool we describe is not limited to particular types of systems.

Deriving documentation from specifications is not new. Leveson, Heimdahl *et al.* [5, 6] describe the provision of full formal requirements specifications for TCAS II avionics systems. Parnas *et al.* [7, 8] describe the documentation of the software design process: '*The validation of design documents* [· · ·] *is a major part of engineering.* [· · ·] *Documentation can be used both as a design medium and as the input to subsequent analysis and testing activities.*' The IDAS project [9] aimed to derive technical documentation in readable natural language, aimed at '*technicians and other experts* [*rather than*] "*the man in the street*" (*e.g., aircraft maintenance manuals, not VCR operations manuals*)'.

In contrast with Leveson *et al.*, we do not aim to document requirements. In contrast with Parnas *et al.*, we do not aim to document a development process for engineering purposes. We do aim to support the provision of a complete, consistent, usable end-user manual for a process control system. We assume the system requirements and development as given. In contrast to the IDAS work, we do not aim to automate the production of natural language instructions, and we are more concerned with documenting real-time systems to enable timely user response in changing situations, such as needed in FCOMs. This does not preclude eventual synthesis between the approaches.

2 Documenting avionics systems

With the introduction of so-called fly-by-wire aeroplanes in commercial transport, avionics now encompasses all aspects of piloting. A pilot of such an aeroplane is only interested in what the specification says about the state variables that are immediately 'to hand', e.g. position of thrust levers and their relation to measured engine power, and its changes. Other variables such as those describing engine fuel-flow controller state, for example, are more important for design and maintenance engineers than for flying the aircraft. Therefore, the pilot's view can be regarded as an abstraction of the design specification of the whole aircraft, i.e. a predicate in a restricted vocabulary of state variables that is logically implied by the design specification.

It is important to distinguish the particular needs of an operator from the general need to document the development and design of a system. An operator will, in general, not care how, or using what process, the system was developed. A pilot only cares how the aeroplane functions right now and how to control it. A pilot flying a landing go-around will use features of the design (position of the controls, for example) in order to command the aircraft to fulfil certain requirements (in this case, performance requirements). You can expect an operations manual to thus contain information from both requirements and design.

We are interested in deriving end-user manuals from requirements or design specifications. The meaning we give to the word 'specification' allows a precise, correct statement of what the operator 'sees' to also be described as a specification. A system may thus have many specifications of this type. (We take it as one of the advantages of the logical approach to specification that it handles this form of abstraction so trivially.) The FCOM is an important end-user document recalling essential information for the pilots and is legally required to be on board an aircraft in flight. It omits information that may be necessary for engineering but is of little help in the cockpit, as well as much information on the system design that pilots would have learnt in a training course. A crucial feature of the FCOM is that the aeroplane (and the pilots) should behave exactly in the way detailed in the FCOM. The FCOM thus should be a specification in our sense, for some part of the total behaviour of the aircraft.

Part of the FCOM for the Airbus A320 aeroplane was analysed by the second author from this point of view as a logical document [3]. The FCOM for a particular A320 [10] was shown to have shortcomings if treated as a rigorous definition, i.e. ambiguities and omissions. The FCOM pages concerned were rewritten as Predicate-Action Diagrams [11], a way of describing finite state machines in which states and transitions between them are labelled with predicates with precise semantics in the Temporal Logic of Actions (TLA). TLA is a temporal logic designed for system specification and verification [12].

What we regard as logical shortcomings in an FCOM may have origins in the conflicting needs of pertinence, coverage and brevity, as well as in simple error. Some shortcomings may arise from a psychological judgment of how best to promote recall of systems and emergency procedures that pilots have learnt in training but that are seldom required. Desiderata for operating manuals may thus conflict. However, we suggest that consistency with specifications is not in conflict with any other desideratum. We call such a manual accurate. (It follows that accurate manuals must be specifications in our sense.) To fulfil other desiderata, a method generating accurate manuals should allow optimisation possibilities according to different criteria. Our method does this, starting from specifications whose accuracy we assume is given.

There is a legal requirement in aviation for accurate flight crew operating manuals. Buck [4] strongly makes the point that manuals frequently fall short of what pilots need in a difficult situation, that pilots in such situations must often improvise, and that this is becoming particularly difficult with the introduction of complex avionics. Avionics systems can be considered interactively complex and tightly-coupled [13]. A suitable illustration may be derived from a recent accident to a Birgen Air Boeing B757 aircraft in February 1996 [14]. There are two relevant features of this accident which illustrate the difficulties of writing easily-usable manuals when the system is interactively complex.

- The captain, who had suffered and identified a failure of his airspeed indicator (ASI), called for the centre autopilot to be turned on. However, the captain's ASI gets its information from the Left Air Data Computer (LADC), as does the centre autopilot. The captain's ASI indicated increasing airspeed with increasing altitude (a phenomenon known to all pilots as consistent with a blocked pilot-static system) and the centre autopilot, following the false airspeed data, raised the nose, causing the aeroplane to slow down and stall. Although there are standard pilot workarounds ('alternate' data paths and a mechanical back-up) which could (you may speculate 'should') have been used, it is hard to imagine that the captain deliberately used an autopilot which he knew to be receiving false data. It could be argued that this is a consequence of interactive complexity: precise information on the centre autopilot/LADC coupling does not appear to be in the operating manual (which must be concise and easily usable, and thus compromise on the amount of information included), but it is included clearly in the much larger systems description document, which is a training and reference manual not intended for quick reference in an emergency.
- The US National Transportation Safety Board (NTSB) suggested an improvement to the operating manual. During the accident sequence, the crew appeared to be confused by two advisory messages, RUDDER RATIO and MACH/SPD TRIM, which appeared on the cockpit Indication and Crew Alerting System (EICAS) display. They are displayed when the stabiliser trim and asymmetry module computer compares the several airspeed sensor inputs and discovers a discrepancy of more than 10 knots. The operations manual apparently did not inform pilots that these two advisories together indicate an erroneous airspeed indication. The NTSB has recommended to the US Federal Aviation Administration that the manual be revised.

Manual writing for an interactively complex system is a task that is not easily optimised. The functionality provided by our simple tool could be expected to help. We adopted a re-engineering approach, respecifying existing manual fragments and applying consistency and completeness checks by hand [13]. This was followed by forward development of the new manual using our tool from these specifications. The contents-in-the-large are determined by the domain experts; we analyse these contents for certain formal properties and preserve those properties during regeneration. The tool we describe here is

- ☐ simple.
- ☐ fully implemented.
- ☐ runs on real, if simple, examples.
- ☐ and produces a manual that is comparable with the original.

Our tool has been used as part of the DiDoLog system [15], a simple experimental forward-generation tool for operating manuals. DiDoLog allows finite state machines to be built graphically using TLA definitions annotated to the states and transitions. TLA+ specifications are automatically generated from the annotated state machines, and finally a logical manual is generated with the tool described in this paper. (DiDoLog requires Tcl/Tk [16].)

We demonstrate the tool by taking the re-engineered logic specification of the A320 braking subsystem [3] and constructing precise FCOM pages from it below. Apart from their precision, these pages are comparable in content and structure to the original. Their layout is also comparable, although we chose to use tables rather than bullet lists (as are used in the FCOM). When our automatic construction techniques are applied to appropriate specifications, the result is also a specification, presented in a more human-oriented way.

This entire paper (rather its electronic version) is itself the result of a single actual run of the tool. The example input is included as source, along with the text body of the paper as meta-comment (see below). The output of running the tool on the source of the paper then yields the text, the example input, plus the output to the examples, formatted as you read below. We did no cut-and-paste on the examples, nor any other post-editing of the paper.

The system is simple, and so are the examples; that is partly our point. Specifications of user interfaces of critical systems are designed to be simple. We are describing a preliminary tool that handles examples of the appropriate order of complexity. However, in the average FCOM, there will be many such examples. Keeping track of the interconnections can be complex without automated help. This is partly why the traditional 'by hand' approach to manual writing is prone to lose rigour.

The usual benefits of automatic generation include the ability to maintain a correct manual under changes of the specification. This aids in collaborative design [17], in which users are involved in assessing designs and hence improving their effectiveness or safety, as practised with human-in-the-loop systems. We have mentioned that our approach facilitates the design-trial-redesign loop. In principle, this could be extended to concurrent engineering, in which technical authors would maintain documentation and evaluate it with operators at the same time as engineers maintain a complete system specification. We have discussed elsewhere techniques to allow technical authors to improve the quality of language used in manuals without compromising the guarantees of automatic generation [18].

3 A language for manuals

Ladkin's preferred FCOM specification is in Predicate-Action Diagrams [3, 14], which have a rigorous formal translation into TLA although use of the temporal-logical operators in that paper is *pro forma*. For this paper it suffices to use Boolean logic.

Our language has Boolean expressions, using `!`, & (or dot), +, => and <=> as the Boolean operators not, and, or, implies, equivalent; others are easy to add. Operator precedence is standard; association is to the left. Two print operators evaluate and print minimised expressions: `>e` prints a minimised equivalent to `e`; and `?e` prints a minimised and factored table in HTML (the hypertext mark-up language used on the World Wide Web [19].

In addition, rewrite rules can be used to simplify expressions. A rule of the form *r*:=*e* allows that the expression *e* is rewritten by *r*. Since rewrite rules may overlap (as in `r:=a&b; q:=b&c; ?a&b&c;`), their application requires minimisation. Use of rewrite rules is illustrated in the examples below.

As you may not wish to print an explanation of something that is trivial, there are two more operators. The slash makes printing using either > or ? conditional; thus >*e*/*p* prints a minimised equivalent of *e* if *p* is true. Secondly, the expression `[]` reduces to `true` iff *e* is a contingency (that is iff *e* is neither strictly true nor false).

The language also provides for name bindings, including functions. To follow the examples below, we mention the following details: comments are taken from % to the end of the line; variable names are either strictly alphabetic strings, or anything between quote symbols (including \" or \\ which represent " and \ in the conventional way).

Our language may not be formally the most elegant or complete. It was implemented to prove a concept, and it works. We do not provide a formal definition in this paper for several reasons: the program is available from the authors, we would rather encourage others to develop our work rather than take it as definitive, and anyway (a point already emphasised) it is a simple system that takes only a week or two to implement. We would like to encourage other workers (particularly in industry) to adopt the concepts, not the specific prototype. The next sections give some examples of real runs with it.

4 Examples

We distinguish `input` and `output` in our examples:

```
% Use a long variable name ···
''This is a simple example using a
function we define'';
This is a simple example using a
function we define
% Define a function equals (one of many
ways to do it)
equals (a, b)
  =a+b=>a & b;
> equals (a, a);
true
> equals (a, true);
a
> equals (equals (a, b), equals (b, a));
true
>''Equals commutes''
  / equals (a, b) <=> equals (b, a);
Equals commutes
```

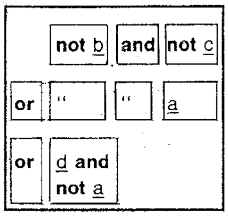
Minimisation becomes more interesting when it is given problems that do not reduce so simply, such as Exercise 1.16.A.3 from Zissos' work [20] on digital circuit design:

```
zissos
  =!a & d+ !b & c & d+a & !b & (c+d)+!b
& !c & !d;
> zissos;
d & !a+!c & !b+!b & a
```

This result is different from the model answer, as the term `! c& !b` is not given: there is a mistake in Zissos' work.

Further, with the automatically generated tabular form, we see that `!b` can be factored out from two terms:

```
? zissos
<- ''Zissos example'';
```



Zissos example

Note how the <– operator specifies a label for the table. The same label is used in the glossary that is automatically generated at the end of this paper (Appendix 12.2).

5 Extracts from the A320 FCOM

Following the simple examples, we now turn to the A320 FCOM. Extensive examples of ambiguities in the FCOM can be found elsewhere [3]. We are content merely to show one example, that an unambiguous section of the FCOM can be duplicated.

Extract from original FCOM:

NORMAL BRAKING
Braking is normal when:
- green hydraulic pressure is available
- A/SKID and N/W STRG switch is ON
- PARKING BRAKE is not ON.

Anti-skid is operative and autobrake is available.

There is additional prose in the FCOM that is descriptive rather than logical. It can be directly copied to the manual generator's output using 'long' variable names, as shown in the examples above, or by using meta-comment.

Appropriate logical variables have been defined [3] and the normal braking condition has been determined to be *normal = green hydraulic and a/s and n/w strg and not park brake and antiskid and autobrake.*

Writing these formulae, together with some explanatory rewrite rules, gives the following manual specification:

```
''both power supply and BSCU
operational'';
  := !(''BSCU failure''+''power supply
failure'')
  -> ''Power normal.'';
''both green and yellow hydraulic
pressure insufficient''
  := !(''yellow hydraulic''+''green
hydraulic'')
  -> ''Hydraulics failure.'';
''nosewheel steering switch is ON''
  := ''n/w strg'';
''nosewheel steering switch is OFF''
  := !''n/w strg'';
% (some specification details omitted
for brevity)
normal
  = ''green hydraulic'' 7 ''a/s'' &
''n/w strg'' & !''park brake'' &
antiskid & autobrake;
% Group lines of input together to get
clearer output ...
(
  > ''<h4>NORMAL BRAKING</h4>'';
  > ''Normal braking mode is achieved
when:'';
  ? normal
    <- ''Normal braking mode''
);
NORMAL BRAKING
Normal braking mode is achieved when:
```

parking brake is OFF **and**
green hydraulic pressure available **and**
antiskid mode active **and**
autobrake armed **and**
antiskid switch is ON **and**
nosewheel steering switch is **ON**

Normal braking mode

We have chosen to display the formula as a table; we believe this to be clear and concise [21]. We could use different styles of presentation. In HTML, the variable names (possibly rewritten) are automatically generated hypertext links to additional explanations of the variables or states being specified.

The tabular representation we chose is a minimal disjunctive normal form (DNF) equivalent to the original formula, with each line in the table being one term from the normal form. This representation closely matches the original form of the FCOM. The DNF lends itself to expressing precisely what the original FCOM chose to represent as bullet lists (one difference noted elsewhere [3] is that some of the bullet lists in the FCOM had obscure semantics).

We implemented the language with a minimisation algorithm that outputs minimised DNF, because it was useful for us. It is also a criterion that can be implemented precisely and optimally, and not confuse our claims with uncertainty over the exact nature of presentation heuristics. Minimal DNF helps, among other things, to check the logical meaning of desired manual entries, and that is what we wanted to try with the A320 manual pages. However, our approach is not beholden to minimal DNF. You can choose other features to minimise. You may prefer to output manual pages which satisfy other structural criteria (for cognitive reasons, for example). We could also generate text that is not best presented as DNF. For example, many manuals contain conjunctions of implications, of the form:

$((s_{11}\ s_{12}.)$ *implies* q_1 *and* $((s_{21}$ *or* $s_{22}\ \ldots)$ *implies* q_2 *and* …

We do not know what users of manuals would like to minimise under what circumstances, although there is evidence (and supporting argument) that minimisation of some form is desirable [22]. For example, would users prefer to minimise the number of terms in a minimal expression, or the maximum number of factors in any term, or perhaps the greatest number of non-trivial common factors? There is scope for experiment, and we would guess that the results would depend on what users (in the current case, pilots) expect of the documentation. We do not wish to impose a particular answer. Indeed, you might provide a set of answers to accommodate various types of intended use.

6 Checking logical properties

Consider the following completeness condition: each state in the system should be documented by some mode in the manual, and modes in the manual should (where appropriate) be mutually exclusive. These and other checks are easy to perform and to document.

In the following example, we first define a function `possible(x)` to be true if `x` is possible, i.e. if `x` is true or is a contingency. Next, if normal mode without antiskid is possible, we output the simple text 'Alternate without antiskid and normal modes overlap when' (i.e. treating it as a simple variable name). Finally, we output as a table the conditions under which normal mode without antiskid is possible. We ran this function as follows:

```
possible(x)
  =x+[x];
normaltwout
  =altwout & normal;
(
  > ''<b> Alternate without antiskid and
normal modes overlap when</b>:'';
  ? normaltwout
)
  / possible(normaltwout)
  <- ''Overlap example''
```

```
Alternate without antiskid and normal
modes verlap when:
```

	parking brake is OFF **and** green hydraulic pressure available **and** antiskid mode active **and** autobrake armed **and** antiskid switch is ON **and** nosewheel steering switch is ON	**and**	BSCU failure
or	"	"	power supply failure

Overlap example

We discover that the FCOM description does not explicitly account for power supply failure or BSCU failure modes. Thus, the tool can be used for completeness checking.

6.1 *Organisation of higher level manuals*

The FCOM is a much reduced form of material included in pilot training, which you would expect to cover items that are omitted or ambiguous in the FCOM. This suggests that you should be able to pick 'higher-level variables' for comment in the FCOM, but also that you should be able to go deeper (simply by adding variables to the 'domain') and view a specification at a more detailed level. This further suggests a hierarchical tool that could operate down to whichever level is desired by the user.

The current approach does not do this; it assumes the specification is 'flat'. However, extension would be easy, such as embedding it within an interactive hypertext system with some theorem proving capability. It would be important for any such extended tool to include various 'completeness' checkers. In particular, for a user to be able to request an abstract manual, the tool should be provided with a suitable summary of what information is not explicitly represented.

6.2 *Glossary*

It is arguable that a manual should contain a glossary and an index of all names used. Our software constructs one with simple hypertext features automatically. We implemented simple features that seemed to us to yield immediate benefits, without making the language unwieldy, but it falls short of what would be required for commercial use. The glossary for the examples in this paper was created automatically by running the source text of this paper through the program.

The glossary appears as follows. Any name, or head token of a rewrite rule, appearing in an explanation is linked to an entry created for it in the glossary. The glossary contains any descriptive text that has been associated with the name or rule, and also links back to the uses of the name. A user of a hypertext version of the manual can click on a phrase occurring in an explanation to obtain a full explanation of that phrase in the glossary, and similarly by clicking in the glossary move back to any use of the phrase in that or other explanations.

Beyond the clear advantages of a complete glossary, our approach closely associates glossary entries with the uses of the names (this is a 'standard' advantage of automatically constructed indexing). Thus, it is easier to check the specification as it is being written, and built-in system checks ensure that all entries in the glossary are annotated.

The language provides three mechanisms to support the hypertext glossary. Any name or rule followed by `->` *string* has *string* appended to its glossary entry. For example:

```
''switch on''
  := sw
  -> ''Mains switch.'';
''switch off''
  := !sw
  -> ''Mains switch.'';
''Mains switch.''
  -> ''The switch is at the bottom of
the red console, pull it forwards to
make it 'on.''';
```

Any expression followed by `<-` *string* is named in the glossary (by the *string*) as the context of use for all of the names in the explanation generated by the expression. As mentioned above, the string also occurs in the explanations generated, as a caption. The sense is that the right-pointing arrow puts text further on in the document, into the glossary, and the left-pointing arrow refers backwards from the glossary to the expression. In the HTML, of course, these forward and backward references are hypertext links.

Finally, as manuals are often structured by sections, the same form without an expression defines a 'global context' for all following explanations, i.e. as if they are in the context of a section with that label. The following example shows both forms in use:

```
<- ''Mains switch example'';
> ''<h4> Mains switch example </h4>'';

Mains switch example
? sw
  <- ''The explanation of sw itself'';
```

switch on

The explanation of sw itself

The glossary for all the examples is at the end of this paper (Appendix 12.2).

7 Guarantees

All good software should come with a guarantee. In many papers, examples are written by hand, or edited from actual program output. Such massaging can lead a reader to wonder if what is presented is an accurate reflection of what the program is really doing. We guarantee that the output the reader sees in this paper is the actual output to the examples. Here's how.

The features of our language so far discussed allow a manual to be constructed from a specification, but do not allow the manual to 'talk about' the specification as an object in itself, which (for example) is what the text of this paper does, and what would be required in various specification documents as a system design is negotiated. Thus, we built a 'meta-mode' for the language.

In normal mode, the program processes all specifications precisely as described in the examples. In meta-mode, however, it processes specifications and additional text. The text is considered as a 'meta'-comment. We have already mentioned the normal '%' comment, and examples of its use have appeared above. Meta-comment has no enforced typographical style, and the comment markers (which are braces) are not represented in the output.

This paper was created by meta-mode. The text of this paper, except the examples, was written as meta-comment. The text was input as 'source', along with the input to all examples as described, but no output to any example. The output is the submitted version of the paper, which should be identical up to typography and line breaks to this version, complete with output and glossary for all the examples. Evaluation in meta-mode enforces the font conventions for distinguishing input and output, to alleviate confusion. The actual source and output is available from both http://www.cs.mdx.ac.uk/harold and http://www.techfak.uni-bielefeld.de/~ladkin/.

Meta-mode is invoked automatically for any specification that uses meta-comments. If an author uses meta-comments in a source, to talk about the specification, the system generates a manual representing those comments, the specification and the explanation. Meta-comments can also be commented out, and even those comments can be meta-commented etc.

8 Implementation issues

TeX [23], LaTeX [24] or SGML [25] would all have been preferable as output mark-up languages. It is no problem to generate them instead: we just find HTML preferable for collaborating over the World Wide Web. HTML's tables (specifically Netscape's tables) are fairly basic and restrict the ways in which a minimal expression can be represented. There are many alternatives that could be tried, such as decision trees, especially if we were to take advantage of interactive representations rather than conventional paper-like forms. Extensions to our approach should permit a choice of representations for the application. In turn, this would permit evaluation of metrics on the output; for example, if it is known that the depth of a decision tree relates to its ease or reliability of use, then the system should summarise such metrics.

The manual specification language was first prototyped as an embedded language within Prolog. Not surprisingly, as minimisation, factorisation and rewriting are all NP-complete optimisation problems, Prolog was too slow for practical use. The program was rewritten in C, and used the standard low-level programming trick of implementing DNF with up to 32 variables as vectors of 32-bit words, which gave it acceptable performance. Minimisation uses the Quine–McCluskey algorithm. There are many alternative algorithms that could also be used (including professional tools for optimising digital logic circuits).

The C program is limited in the complexity of queries it can handle (because the minterm algorithm can generate huge numbers of minterms). This has not been a practical limitation; you could argue that use of a specification that required more than 32 Mb of RAM could involve such a degree of complexity that a mechanically generated manual would not help much.

9 Conclusions

We have given a proof of concept that a user manual can be produced automatically, using straightforward programming, in such a way as to be comparable with a desired original (modulo typography). This technology shows how easy it is for manual writers to check completeness and correctness, and indicates that, modulo the goals of particular manuals, automatic production of correct manuals is a relatively straightforward task. We have highlighted that the speed-up and quality improvements enable concurrent design, which should result in better systems through involving operators earlier, and more effectively, in the design process.

We have not done research into the 'best' form of FCOM, one meeting all the desiderata in mutually optimal ways. Such questions would need to be addressed in any development of our work. We have shown here that we could provide the FCOM material with logic and precision, by using an automatic tool. To do that we did not need to address the human factors questions. Our tool, particularly given its precise functionality, can be used in controlled experimentation.

10 Acknowledgments

The authors would like to thank Hagen Barth who ported the program and identified and fixed some hardware/

compiler dependencies; and Paul Curzon and the referees, who made very useful comments on the paper.

11 References

[1] SWARTOUT, W., and BALZER, R.: 'The inevitable intertwining of specification and implementation', *Commun. ACM*, 1982, **25,** (7), pp. 438–440

[2] BOEHM, B.W.: 'A spiral model of software development and enhancement', *ACM SIGSOFT Softw. Eng. Notes*, 1986, **11,** (4), pp. 14–24

[3] LADKIN, P.B.: 'Analysis of a technical description of the Airbus A320 braking systems', *High Integrity Syst.*, 1995, **1,** (4), pp. 331–349 (also available from http://www.techfak.uni-bielefeld.de/~ladkin)

[4] BUCK, R.N.: 'The pilot's burden: flight safety and the roots of pilot error' (Iowa State University Press, 1994)

[5] LEVESON, N., HEIMDAHL, M.P.E., HILDRETH, H., and REESE, J.D.: 'Requirements specificaton for process-control systems', *IEEE Trans.*, 1994, **SE-20,** (9), pp. 684–707

[6] HEIMDAHL, M.P.E., and LEVESON, N.: 'Completeness and consistency analysis of state-based requirements', *IEEE Trans.*, 1996, **SE22,** (6), pp. 363–377

[7] PARNAS, D.L., MADEY, J., and IGLEWSKI, M.: 'Precise documentation of well-structured programs', *IEEE Trans.*, 1994, **SE-20,** (12), pp. 948–976

[8] PARNAS, D.L., and MADEY, J.: 'Functional documents for computer systems', *Sci. Comput. Program.*, 1995, **25,** pp. 41–61

[9] REITER, E., MELLISH, C., and LEVINE, J.: 'Automatic generation of technical documentation', *Appl. Artif. Intell.*, 1995, **9,** (3), pp. 259–287 (also available from http://www.dai.ed.ac.uk/daidb/staff/personal_pages/chrism/idas.html)

[10] Main Commission Aircraft Accident Investigation Warsaw: 'Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993'. Warsaw, March 1994 (also available from http://www.techfak.uni-bielefeld.de/~ladkin)

[11] LAMPORT, L.: 'TLA in pictures', *IEEE Trans.*, 1995, **21,** (9), pp. 768–775 (also available from http://www.research.digital.com/SRC/tla)

[12] LAMPORT, L.: 'The temporal logic of actions', *ACM Trans.*, 1994, **16,** (3), pp. 872–923 (also available from http://www.research.digital.com/SRC/tla)

[13] PERROW, C.: 'Normal accidents: living with high-risk technologies' (Basic Books, 1984)

[14] LADKIN. P.B. (Ed.): 'Computer-related incidents and accidents with commercial airplanes', *Hypertext Commendium of Sources and Commentary* (also available from http://www.techfak.uni-bielefeld.de/~ladkin)

[15] BARTH, H.: 'DiDoLog: automatable generation of specifications and formally correct manuals from informal sources'. Diploma Master's Thesis, Technische Fakultät, Universität Bielefeld (also available from http///www.techfak.uni.bielefeld.de/ladkin)

[16] OUSTERHOUT, J.K.: 'Tcl and the Tk toolkit' (Addison-Wesley, 1994)

[17] THIMBLEBY, H.W.: 'Creating user manuals for use in collaborative design'. ACM Conf. on Computer-Human Interaction, CHI'96, Vancouver, Canada, 1996, pp. 279–280

[18] THIMBLEBY, H., and LADKIN, P.B.: 'A proper explanation when you need one' *in* KIRBY, M.A.R., DIX, A.J., and FINLAY, J.E. (Eds.): 'People and Computers X'. Proc. BCS Conf. on HCI'95 (Cambridge University Press, 1995), pp. 107–118 (also available from http./www.techfak.uni-bielefeld.de/~ladkin)

[19] NCSA: 'A beginner's guide to HTML' (available from http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html, 1996)

[20] ZISSOS, D.: 'Logic design algorithms' (Harwell/Oxford University Press, 1972)

[21] OKUNO, H., MATSUMOTO, H., and ASAI, H.: 'TableSpec: free format specification table and source code generation', *Softw.—Pract. Exp.*, 1996, **26,** (2), pp. 213–235

[22] CARROLL, J.M.: 'The Nurnberg funnel: designing minimalist instruction for practical computer skill' (MIT Press, 1990)

[23] KNUTH, D.E.: 'The TeXbook' (Addison-Wesley, 1992)

[24] LAMPORT, L.: 'LaTeX: a document preparation system' (Addison-Wesley, 1994)

[25] BRYAN, M.: 'SGML: an author's guide' (Addison-Wesley, 1988)

12 Appendix

12.1 *HTML comments*

Although HTML is generated, the system does not interpret HTML itself. Thus, HTML tags may be used in variable names and in meta-comments. These tags have side-effects on the mark-up of the manual. One use of this is defining variables with names that are HTML section headings, as in some of the examples. This feature was also exploited in this paper to omit some detail not relevant for our exposition.

We have noted that details of the specification were omitted. This was done by enclosing the specification between HTML start and end comment tags, themselves enclosed inside meta-comment braces, as follows:

```
% material omitted... (this is a comment
that does appear)
{< !--}
text that is processed, but does not
appear in the paper
{-->}
```

Thus, the program processes the complete specification, yet allows us to present a briefer paper uncluttered with the detail in the specification needed for completeness with respect to the original.

A production system would be likely to control such freedom of expression closely. Consider that, although masquerading in the description above as a processed line of text ("`text that is processed ...`"), in fact it was not processed.

The prototype provides a very coarse control switch. It is possible to generate a manual with either full copying of input (as in this paper) or with no copying. The former is useful for working with a specification when seeing its manual; the latter is useful for using a manual without its logic specification intruding. The switch then allows a single file to be used for both development/debugging and use, hence further helping to reduce errors in manuals.

12.2 *Glossary*

Note that underlining indicates hypertext links

a
Used in:
Zissos example

antiskid mode active
Used in:

Normal braking mode
Overlap example
Specification:
antiskid
antiskid switch is ON
Used in:
Normal braking mode
Overlap example
Specification:
a/s
autobrake armed
Used in:
Normal braking mode
Overlap example
Specification:
autobrake
b
Used in:
Zissos example
BSCU failure
The double channel Brake Steering Control Unit (BSCU) controls the anti-skid system via the alternate servo valves.
Used in:
Overlap example
c
Used in:
Zissos example
d
Used in:
Zissos example
green hydraulic pressure available
NB, check yellow hydraulic available too.
Used in:
Normal braking mode
Overlap example
Specification:
green hydraulic
nosewheel steering switch is ON
Used in:
Normal braking mode
Overlap example
Specification:
n/w strg
parking brake is OFF
Used in:
Normal braking mode
Overlap example
Specification:
not park brake
power supply failure
Used in:
Overlap example
switch on
Mains switch. The switch is at the bottom of the red console, pull it forwards to make it 'on'.
Used in:
The explanation of sw itself *in* Mains switch example
Specification:
sw

The paper was first received 9 April and in revised form 11 September 1996.

Harold Thimbleby is with the School of Computing Science, Middlesex University, London N11 2NQ, UK; Peter Ladkin is with Technische Fakultät, Universität Bielefeld, Bielefeld, D-33501, Germany.