The effects of structure on the comprehensibility of formal specifications

Finney KM
Fedorec AM
Fenton NE

9 May, 1997

Abstract

Use of structuring mechanisms (such as modularisation) is widely believed to be one of the key ways to improve software quality. Structuring is considered to be at least as important for specification documents as for source code, since it is assumed to improve comprehensability. Yet, as with most widely-held assumptions in software engineering, there is little empirical evidence to support this hypothesis. Also, even if structuring can be shown to be a good thing, we do not know how *much* structuring is somehow optimal. One of the more popular formal specification languages, Z, encourages structuring through its schema calculus. We describe a controlled experiment in which we tested two hypotheses about the effects of structure on the comprehensibility of Z specifications. We found evidence that structuring a specification into schemas of about 20 lines long significantly improved comprehensability over a monolithic specification. However, there seems to be no perceived advantage in breaking down the schemas into much smaller components. Our experiment is fully reproducible.

1. Introduction

One of the most widely accepted concepts in software engineering is that structure (and more specifically modularisation and information hiding) is a key to controlling complexity and thereby improving quality. Indeed, [Fenton and Pfleeger 1996] describes how 35 years of methodological research in software engineering has produced little more than paradigms to support this single concept. These range from structured programming and structured design and analysis through to object oriented design and abstract data typing.

Although everyone seems to accept the hypothesis that structuring is a good thing there is surprisingly little empirical evidence to show that it leads to genuine quality improvements. This is true even for such widely used methods as structured programming [Fenton et al 1994]. The evidence to support modularisation at the programming level is in fact especially weak. We refer to modularisation in the most general sense: decomposing a complex object into smaller (but still coherent) objects. In the small number of empirical studies of industrial programs the evidence actually suggests that larger modules tend to have lower fault rates [Basili and Perricone 1981, Hatton 1995, Moeller and Paulish 1995].

Despite this disappointing empirical evidence, the intellectual arguments in favour of structuring are very strong if we consider improved comprehensability as a key success criteria. This should be especially the case for writing specifications. However, even if it can be shown that structuring leads to improved comprehensibility it is not at all obvious just what constitutes the ‘right’ level of structurendess. In this paper we describe an experiment whose objectives were to test hypotheses about the impact of structure on comprehensability for one of the more popular formal specification languages, Z [Spivey 1992]. By its nature Z is written in sections called schemas which can vary in length and complexity. In a manner parallel to the use of modules, procedures and subroutines within programming languages, it is also possible for Z users to design a specification using different structural characteristics. It is possible to write a specification using a single monolithic schema. Alternatively, the specification may be broken down into very small components so that there are numerous individual schemas dealing with single operations, functions or state changes. Our main result is that there is evidence that structuring a specification into schemas of about 20 lines long significantly improves comprehensability over a monolithic specification. In other words we are able to confirm the widely accepted (but previously untested) hypothesis. However we also show that there seems to be no perceived advantage in breaking down the schemas into much smaller components.

In Section 2 we describe the previous empirical studies which provide the most relevant background to the current study. None specifically tested the effect of structuring on the comprehensibility of formal specifications. However, we are able to report on empirical studies which fall into three other relevant categories:

1. studies that have looked at the effect of structuring on the comprehensibility of programs
2. studies that have used rigorous approaches to measure comprehensibility (such studies might be looking at factors other than structure as the dependent variable)
3. studies that have looked at factors affecting the comprehensibility of formal specifications (or more specifically, Z specifications).

In Section 3 we describe the design of our own experiment, while in Section 4 we present the results.

2. Previous relevant empirical studies

Our objective in this and previous work has been to assess empirically the comprehensibility of formal specifications and the factors which impact on it. We believe such work is important because proponents of formal methods have tended to underestimate the difficulty that users have in understanding even small formal specification documents. For example, while the following claim:

‘After a week’s training in formal specification, engineers can use it in their work’ [ConForm 1997]

is typical of many that have been made, the little real evidence that exists suggests that students of formal methods find it very hard to understand even the simplest specifications after some weeks of training [Finney 1996]. The problem seems to be that proponents of formal specification notations have tended to be drawn from those computer scientists with a mathematical background. Their own experience with logic, set theory and the interpretation of symbolic language does not always give them an appreciation of the difficulties experienced by other users when reading these notations. One of the most widespread formal specification notations, Z [Spivey 1992], has been claimed to be more easily comprehensible than other formal notations because of its use of schemas and stylised layout. Significant benefits are claimed for the use of Z in requirements specification because of improved communication between development teams and between specifiers and clients, [Barden and Stepney 1992]. However, the underlying notation is still based on extensions of set theory and logic and also incorporates some of its own specialised symbols.

Despite the widespread assumption that structuring formal specifications leads to improved comprehensibility we have found no previous empirical studies that have directly tested this hypothesis. Nevertheless there are a number of important empirical studies which are relevant in other respects. In this section we review the most relevant ones..

2.1 Studies that have looked at the effect of structuring on the comprehensibility of programs

There have been many studies which have examined the efficacy of structured programming and specific programming constructs ([Vessey and Weber 1984] reviewed these studies and concluded that, despite the compelling intellectual arguments, there was no equivocal evidence to support structured programming). There have also been a number of studies that have examined the effect of structuring on fault rates ([Hatton 1997 reviews these and shows, curiously that the number of faults per line of code tended to decrease, rather than increase, with module size. However, there have been remarkably few studies that have specifically examined the effect of structuring mechanisms on program comprehensability. One of the early studies by Parnas provided some weak evidence that decomposition into a modular structure helped improve program comprehensability [Parnas 1979]; he felt that the particular way the decomposition was done was critical. After giving two different decompositions, he stated that because of the method of decomposition used only one of them makes sense when viewed as a whole.
<KATE: ADD THE TENNY STUFF - I NO LONGER HAVE YOUR CHAPTER 5!>In his studies of Cobol programmers [Harold 1986] tried experimental evaluation of program quality and included features of 'readability and understandability'. (). As factors that might affect these two aspects of a program he included comments, procedure names, sequential flow of logic, module size, indentation and logical simplicity. He found evidence to link the quality of programmes to the techniques of structuring; however the sample size of 20 was quite small and some of the results needed further evaluation.

2.2 Studies that have used rigorous approaches to measure comprehensibility

[Takang et al 1996] stresses the need for proper statistical techniques to analyse the data from experiments on program comprehension data. Takang and his colleagues conducted an experiment using 89 students and used both an objective and subjective means of assessing comprehensibility. Three hypotheses were tested: 1) That commented programs were more understandable than those without; 2) That programs with full identifier names were more understandable than those without; 3) The combined effect of comments and full identifier names was better than either independently. Only hypothesis 1 was supported by the objective scores but only hypothesis 2 was supported by the subjective scores.

2.3 Studies that have looked at factors affecting the comprehensibility of formal specifications

Until recently very little empirical work has taken place in the area of comprehensibility applied to formal specifications. In a set of technical reports [Vitner et al, Vitner and Loomes, Vitner 1996] Vitner looks at the psychology of reasoning about logical statements and does pilot tests on 12 subjects using logic and Z. He then extends this work with two further experiments making comparisons using computer scientists, with 60 and 40 subjects respectively. These were found in several different institutions and took part by answering a series of questions sent by post. The conclusions seem to be from this work so far, that there are errors in reasoning about logic arising from the use of Z which are similar to those found when using the same information expressed in natural language.

[Mitchell et al. 1994] looked at the effects of restructuring an existing specification written in Z by Hayes [Hayes 1987]. They took the original specification and rewrote it with different structural characteristics. They

concentrated on separating the essential functionality of the system from the part of the specification dealing with the presentation of that functionality to the users.
They concluded that in their reorganisation there was a trade off in benefits. In their new specification the overall view of the system was easier to understand whereas understanding the component parts was more difficult; the reverse was true in the Hayes original.

[Finney et al 1997] carried out experimental work to investigate the effects on the comprehensibility of Z notation due to changing variable names and commenting. They found that meaningful naming of variables had an effect on the ability of subjects to understand the specification. They also found that the presence of natural language comments were predictive of a good level of understanding but did not by themselves ensure comprehensibility.

While the above studies provide insight about relevant experimentation and measurement, none have addressed directly the impact of structure on formal specifications. However, during our experimentation in [Finney 1997] there was a response from the subjects that the structure of the Z specification affected their ability to read and understand it. Thus the current paper extends the work of [Finney 1997] study looking at the factors which may affect the comprehensibility of a Z specification. Two of the authors have already .. From this studyby describing a new experiment to look at how the structural decomposition alone effects the comprehensibility.

3. The experimental design

In this experiment we were trying to assess whether comprehensibility is affected by structure in a formal Z specification. This implies that structure is the independent variable we can control and that we must find a way of assessing the dependant variable; i.e. choosing a suitable measure for comprehensibility. To control the independent variable three levels of structuring were chosen to give a reasonable variation in the style of Z. We based our measurements of comprehensability on the approach proposed by Brooks in his paper about experimental studies of programmers' behaviour [Brooks 1980]. We asked 20 questions with the total score obtained as the measure of the dependant variable.

The first (null) hypothesis being tested test was;

Hypothesis 1: comprehensibility is not improved as a result of the modularisation of a Z specification

To test this the specification of a small business telephone directory designed to deal with employees' telephone numbers and office location was written. It was constructed in three ways:

- Specification A was completely unmodularised consisting of 121 lines of which lines 12 - 121 were a single monolithic Z schema. (See Appendix 1 for sample page).
- Specification B was modularised; it grouped the same information into 6 main schemas consisting of 3 schemas dealing with operations and 3 matching sets dealing with error handling. Each of the schemas was about 20 lines long and there were 159 lines in all as some repetition was incurred by this process (See Appendix 2 for sample page).
- Specification C, was highly modularised; this took 18 smaller schemas to convey the same information. It was the longest specification with 165 lines,. (See Appendix 3 for sample page).

All three specifications were functionally identical. Great care was taken to make commenting and notation match in all three so that each reader was given the same information. The only difference between them was in the structural forms.

Thesecond hypothesis being tested was:

Hypothesis 2: comprehensibility is not improved by reducing the size of the modules

This hypothesis was tested by looking at the differences between the results of using specifications B and C.

Twenty questions were designed to test the comprehensability of the specification in a variety of ways. (See Appendix 4 for a sample). All lines of the specification were numbered so that reference could be made to particular parts of the specification in the questions and answers. Broadly the questions fell into four categories, testing the subjects' competence at:

- Finding a relevant part of the specification
  e.g. Which line in the specification tells you......
- Understanding the notation e.g. What is indicated by the difference in number? and number! in line ......
- Relating the specification to the model
  e.g. Describe the purpose of line.......
- Modifying the specification by writing an extra feature
  e.g. Add the lines needed to include.........

The subjects of the experiment were 65 computing students who were just finishing a one semester course in Z. Their tuition time was approximately 40 hours. This compares favourably with the training time normally given to software engineers in industry to learn the notation. Thus the fact that the subjects here were students, rather than professionals, does not compromise the validity of the experiment. The students were divided into three groups with each group being subjected to exactly one specification. To ensure that the spread of abilities was the same within each specification the students were ranked using the average mark obtained in the eight units of their previous year of study.

4. The experiment

At the time of the experiment each student was given a named pack containing the relelvant specification, a question sheet and an answer template. They were each given an hour to attempt the 20 questions.
All marking was done by the same person (one of the authors) Scores were recorded for each individual question and a total out of 60 was awarded.

There was some imbalance in the groups due to students not attending on the day. The final numbers in each group were

Specification A	23
Specification B	23
Specification C	19

4.1 The results

From a possible total of 60, the scores varied from the highest at 47 to the lowest at 7. The results (sorted in ascending order) are shown in Table 1 and Figure 1. An initial inspection shows specification A clearly with lower overall scores than the other two.

SPEC A		7	8	10	11	13	14	17	17	18	18	19	19	19	21	23	23	24	28	28	29	30	35	41
SPEC B		9	14	15	15	19	21	22	23	24	25	26	27	27	29	29	31	32	34	35	35	35	39	47
SPEC C		10	16	19	19	19	19	23	25	25	26	28	28	29	32	38	40	40	41	45				

The average scores were	Specification A	20.52
	Specification B	26.65
	Specification C	27.47

Table 1: Scores sorted in ascending order

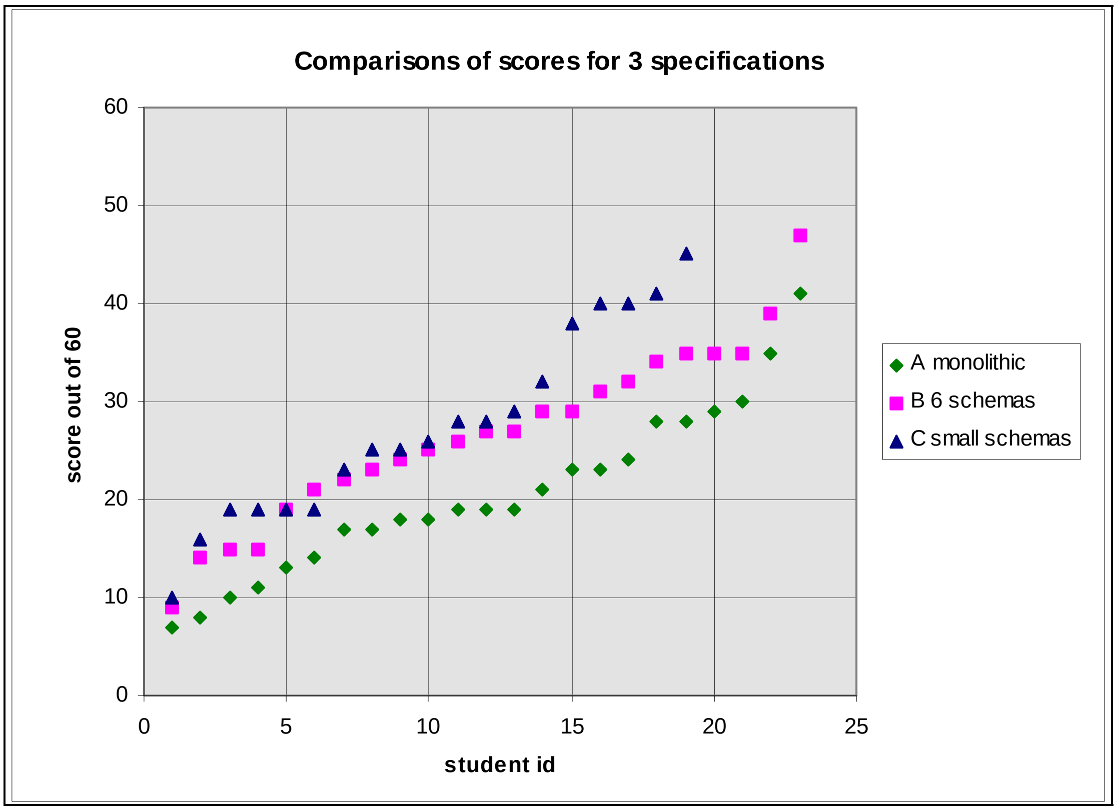

Fig 1 Comparison of student scores for each Specification

From Figure 1 we can see that the scores obtained for specification A (the unmodularised schema) were lower overall than the others but that the differences were not so marked at either end of the scoring range. From Table 1 we see that specification A has 5 scores below 14 in comparison with 1 each for specifications B and C.

This might suggest that even a basic level of information was hard to interpret from the unodulrarised specification. At the other end of the range Specification C with many small schemas gives rise to 5 marks over 35 in comparison with 2 for Specification B and for Specification A. This suggests that better comprehension occurs with very small schemas. However it must be stressed that these are merely impressions gained from looking at the raw data and the effect of ranked order can distort the way comparisons are made.

4.2 Statistical analysis

The first aim was to see if there was a significant difference between the three specifications, bearing in mind the large variation in scores within each treatment. A one way independent measures analysis of variance (ANOVA) was applied to the data using the Minitab statistical package [Minitab] and Table 2 shows the output.

One-Way Analysis of Variance

```
Source   DF      SS      MS      F      P
 Data     2   627.7   313.9   3.86  0.026
Error    62  5042.1    81.3
Total    64  5669.8
```

Table 2 The Output from the ANOVA using Minitab

The value of p = 0.026 indicates a significant difference between the three specifications. Henceit is legitimate to look at some pairwise comparisons between the different specifications, based on the data summarised in Table 3.

Specification	No. of Values	Mean	Standard Deviation	Standard Error of mean
A (Monolithic)	23	20.52	8.48	1.8
B (6 schemas)	23	26.57	8.95	1.9
C (small schemas)	19	20.9	9.71	2.2

Table 3 Comparisons between different specifications.

Using a two sample t -test we obtain the results for the three pair-wise comparisons that at the 5% level

A and B   p = 0.023 with a 95% confidence interval for the difference of the means A -B [-11.2, -0.9]
A and C   p = 0.020 with a 95% confidence interval for the difference of the means A-C [-12.7, -1.2]
B and C   p = 0.76   with a 95% confidence interval for the difference of the means B-C [-6.8, 5.0]

By applying these pairwise tests we can confirm that there are significant differences between A and B and also A and C. There is no significant difference between B and C.

With three comparisons the danger of using a t-test is in introducing an unacceptable level of Type I error, that is we may reject the null hypothesis when it is true. Because of this potential weakness of the t-test we also applied two more stringents tests, namely Tukeys Honestly Significant Difference (HSD) and the Scheffé test (see Appendix 5). In both cases we found confirmation of significant differences between A and B and also between A and C (but not between B and C).

5. Conclusions and Further work

We have described a fully reproducible experiment to examine the effects of modularisation on the comprehensability of Z specifications. We tested two hypotheses:

Hypothesis 1: comprehensibility is not improved as a result of the modularisation of a Z specification

Hypothesis 2: comprehensibility is not improved by reducing the size of the modules
The results of the experiment provide evidence for rejecting hypothesis 1. In other words we found evidence that comprehensibility is improved as a result of the modularisation of a Z specification. Although software engineering experts may find this result 'self-evident' it is nevertheless the first rigorous empirical evidence of it. Moreover, as discussed in Section 2, similar 'self-evident' hypotheses about the modularisation of source code has not been supported by empirical evidence.
We can conclude that, in writing a formal specification, some care should be taken over the structural presentation. Clearly one large block of Z makes comprehensibility difficult but there seems to be no perceived advantage in breaking down the schemas further once they are about 20 lines long. Specifically our experiment found no evidence for rejecting hypothesis 2 (that highly modularised specifications do not improve comprehensability over modularised ones).

As with all controlled experiments of this nature there are limitations which mitigate against wide generalisation of the results. One limitation is the size of the problem being specified. Although larger than those typically used in many student-based experiments, the specifications here could not be described as representing industrial-sized systems. Hence it is not clear that the results 'scale-up'. However, it must be stressed, that popular practice dictates that even very large systems should be broken down into subsystems of the kind of size considered here. Another limitation is that we have considered only a single factor affecting comprehensability. [Tenny ?], in his study of programming languages, argued that structuring may not be the best way to improve comprehension; simply adding comments may be better. It could well be the case that, if presented with a monolithic Z specification, its comprehensability could be most cost-effectively improved by adding comments rather than attempting some modularisation. This kind of joint-effects issue was not considered in our experiment. However, this is not a serious limitation because we have shown that, irrespective of other factors, a *starting* strategy of modularisation is preferable to one of non-modularisation if comprehensability is an issue.

The most commonly cited limitation of student based experiments in software engineering is that performance of students on classroom problems cannot possibly be used to make inferences about performance of professionals on industrial-scale problems. We believe strongly that this limitation does *not* apply in this case. The amount of Z training our students received compares very favourably to that given to professional software engineers. Indeed, there are no significant numbers of professionals who have greater experience of using Z than these students. On the contrary, the proponents of formal methods are keen to push the idea that software engineers can start to write and read Z specifications after just one week's training. In that case our results are directly relevant to the typical software engineers experiencing Z specifications; it would actually be of less relevance if this experiment had been carried out on the (small number of) software professionals who can accuratley be regarded as 'experienced Z users'.

We invite other researchers to replicate our experiment. Our own plans for improving the experiment include:

- using larger specifications to allow analysis of a greater variety of schema size so that more guidance over size would be possible
- investigating the correlation between students' known abilities as demonstrated by their previous marks, and their ability to read and comprehend Z. After some initial analysis these two statistics seem strongly

correlated. This might lead to implications for aspects of previous experience which enable software engineers to use formal methods successfully

- analysis of the scores for different types of question that could lead to conclusions about the difference between reading and understanding a specification
- including timing. It may be that the physical separation of a large number of small schemas within a document would add considerably to the time taken to read a specification when compared with a more dense but compact structure

.

.
.

References

Barden R and Stepney S, Support for using Z. Z User Workshop (Eds Bowen JP and Nicholls JE) Springer-Verlag, London 1992

Basili VR and Perricone BT, Software Errors and Complexity: An Empirical Investigation, Communications of the ACM 27(1), pp. 42-52, 1984.

Brooks RE Studying Programmer Behaviour Exprerimentally: The Problems of Proper Methodology Communications of the ACM, 23 (4), pp.207 - 213, April, 1980

Fenton NE and Pfleeger SL, Software Metrics: A Rigorous and Practical Approach (2nd Edition), International Thomson Computer Press, 1996.

Fenton NE, Pfleeger SL, Glass R, Science and Substance: A Challenge to Software Engineers, IEEE Software, 11(4), 86-95, July, 1994.

Finney K, Rennolls K and Fedorec A Measuring the Comprehensibility of Z Specifications Acceptedfor publication Feb 1997 to be published in JSS 1998

Harold, F.G., Experimental Evaluation of Program Quality Using External Metrics Empirical Studies of Programmers eds Soloway. E., Iyengar. S., Ablex Publishing Corp NJ 1986

Hatton L Logarithmic disorder in software systems: evidence and implications, Preprint, 1995.

Hatton L Reexamining the Fault Density-Component size Connection, IEEE Software, 14 (2), pp. 89-97, March, 1997.

Hayes.I.,Specification Case Studies Prentice Hall 1987

Hinton P.R, Statistics Explained, Routledge 1995

Minitab reference manual. Minitab Inc.3081 Enterprise Drive, State College PA 16801 USA

Mitchell. R., Loomes. M., and Howse. J., Structuring formal Specifications - a lesson relearned Microprocessors and Microsystems 18 (10) 593-599   (1994)

Moller K-H and Paulish D, An empirical investigation of software fault distribution, in 'Software Quality Assurance and Measurement' (Eds Fenton NE, Whitty RW, Iizuka Y), International Thomson Computer Press, 242-253, 1995.

Parnas, D., On the criteria to be used in Decomposing Systems into Modules Classics in Software Engineering (Ed Edward Yourdon) Yourdon Press New York 1979

Spivey. J..M., 'The Z Notation: A reference Manual'. 2nd edition Prentice-Hall International 1992

Takang AA, Grubb PA, MaCredie RD, The effects of comments and identifier names on program comprehensibility: an experimental investigation, Journal of Programming Languages 4, 143-167, 1996.

Vinter, R.J., A Review of Twenty Four Formal Specification Notations ,University of Hertfordshire, Division of Computer Science Technical Report 240 Feb 1996, 1996.

Vinter, R. J. , Loomes, M.J. and Kornbrot, D.E., Reasoning about Formal Software Specifications: An Initial Investigation ,University of Hertfordshire, Division of Computer Science Technical Report 249 March 1996, 1996.

Vinter, R. J., Transfer of Logical Tendances to Formal Reasoning ,University of Hertfordshire, Division of Computer Science Technical Report 252 July 1996, 1996.

Loomes, M.J. and Vinter, R. J., Formal Methods: no cure for Faulty Reasoning ,University of Hertfordshire, Division of Computer Science Technical Report 265 Sep 1996, 1996.

Appendix 1
Part of the monolithic Specification A

This schema includes setting up an empty phone directory, plus all operations and the errors arising from the failure of the operations: AddPerson, DeletePerson**,** QueryPersonNumber, QueryPersonOffice**,** ListNumberPeople, ListOfficePeople, ListOfficeNumbers, MovePerson**,** MovePhone**,** ConnectPhone and DisconnectPhone.

Note as both Moving operations use the override function we need not produce error conditions it will be the trivial override.

```
12      é PhoneOperations èèèèèèèè
13      æ  DPhoneDir
14      æ  name? : NAME
15      æ  office?, office! : OFFICE
16      æ  number?, number! : NUMBER
17      æ  operation? : OPERATION
18      æ  resp! : RESPONSE
19      æ  people! : P NAME
20      æ  phones! : P NUMBER
        çèèèèèèèè
21      æ  (    resp! = ok
22      æ  (    (    operation? = InitPhoneDir
23      æ            addr´ = phone´ = location´ = Æ
24      æ       )  Ú
25      æ       (    operation? = AddPerson
26      æ            name? Ï dom phone
27      æ            phone’ = phone È {name? ª number?}
28      æ            addr’ = addr È { name? ª office? }
29      æ            (    (    number? Ï dom location
30      æ                      location´ = location È { number? ª office? }
31      æ                 )  Ú
32      æ                 (    number? ∈ dom location
33      æ                      location number? = office?
34      æ                      location’ = location
35      æ       )    )    )  Ú
36      æ       (    name? Îdom phone
37      æ                 (    operation? = MovePerson
38      æ                 phone’ = phone Å {name? ª number?}
39      æ                 addr’ = addr Å { name? ª office? }
40      æ                 location´ = location Å { number? ª office? }
41      æ            )  Ú
42      æ                 (    (    operation? = DeletePerson
43      æ                      phone’ = phone \ {name? ª phone name?}
```

Appendix 2

The operations described here on the phone directory are:

- *AddPerson* **- Assign a person a phone and an office.**
- *DeletePerson* **- Remove a person from the directory.**

```
16      é BasicOperations èèèèèèèè
17      æ  DPhoneDir
18      æ  name? : NAME
19      æ  office? : OFFICE
20      æ  number? : NUMBER
21      æ  operation? : UPDATE
22      æ  resp! : RESPONSE
        çèèèèèèèè
23      æ  (    operation? = AddPerson
24      æ       name? Ï dom phone
25      æ       phone' = phone È {name? ª number?}
26      æ       addr' = addr È { name? ª office? }
27      æ       (    (    number? Ï dom location
28      æ                 location´ = location È { number? ª office? }
29      æ            ) Ú
30      æ            (    number? ∈ dom location
31      æ                 location number? = office?
32      æ                 location´ = location
33      æ  )    )    ) Ú
34      æ  (    operation? = DeletePerson
35      æ       name? ∈ dom phone
36      æ       phone' = phone \ {name? ª phone name?}
37      æ       addr' = addr \ { name? ª addr name? }
38      æ  )
39      æ  resp! = ok
        êèèèèèèèèèèèèèèèè
```

Appendix 3

Several of the small schemas used in Specification C

The operation described here on the phone directory is:

AddPerson - **Assign a person a phone and an office.**

```
12 é AddPerson èèèèèèèè
13 æ  DPhoneDir
14 æ  name? : NAME
15 æ  office? : OFFICE
16 æ  number? : NUMBER
17 æ  resp! : RESPONSE
   çèèèèèèèè
18 æ  name? Ï dom phone
19 æ  (    (    number? Ï dom location
20 æ            location´ = location È
21 æ                 { number? ª office? }
22 æ       )  Ú
23 æ       (    number? ∈ dom location
24 æ            location number? = office?
25 æ            location´ = location
26 æ  )    )  Ù
27 æ  phone' = phone È {name? ª number?}
28 æ  addr' = addr È { name? ª office? }
29 æ  resp! = ok
   êèèèèèèèèèèèèèèè
```

This schema shows the errors arising from the failure of the operation *AddPerson*

```
30 é AddFailures èèèèèè
31 æ  K PhoneDir
32 æ  name? : NAME
33 æ  resp! : RESPONSE
34 æ  number? : NUMBER
35 æ  office? : OFFICE
   çèèèèèèèè
36 æ  (    name? ∈ dom phone
37 æ       resp! = adderror
38 æ  )  ∨
39 æ  (    number? ∈ dom location
40 æ       location number? ≠ office?
41 æ       resp! = connecterror
42 æ  )
   êèèèèèèèèèèèè
```

The operation described here on the phone directory is:

DeletePerson - **Remove a person from the directory.**

```
43 é DeletePerson èèèèèèèè
44 æ  DPhoneDir
45 æ  name? : NAME
46 æ  resp! : RESPONSE
   çèèèèèèèè
47 æ  name? ∈ dom phone
48 æ  phone' = phone \ {name? ª phone name? }
49 æ  addr' = addr \ { name? ª addr name? }
50 æ  location' = location
51 æ  resp! = ok
   êèèèèèèèèèèèèèèè
```

This schema shows the errors arising from the failure of the operation *DeletePerson*

```
52 é DeleteFailures èèèèèèè
53 æ  K PhoneDir
54 æ  name? : NAME
55 æ  resp! : RESPONSE
   çèèèèèèèè
56 æ  name? Ï dom phone
57 æ  resp! = connecterror
   êèèèèèèèèèèèèèè
```

Appendix 4

An example of the set of 20 questions to accompany Specification A

1. What information is contained in the line numbered '1' of the specification?
2. What is indicated by the difference between *number*? and *number*! in line 16?
3. What information is given in line 28 of the specification?
4. Describe the purpose of line 68.
5. Show how lines 27 and 28 would appear if a new person, 'Jones', is to be given office number '48B' and a phone number '376'.
6. List all the lines of the specification that would need to be changed if it was decided to rename the function *addr*?
7. Describe the condition(s) which give rise to the response '*adderror*'.
8. Why is the type of '*phones*!' in line 20 $\mathbb{P}$*NUMBER* and not just *NUMBER*?
9. Which 5 conditions in the telephone system give rise to the response '*notfound*'?
10. Explain the significance of the '⊕' symbol in line 89.
11. Which line informs you that a phone remains in the same office when a person is removed from the directory?
12. Which line in the specification tells you that a person can only have one phone?
13. Which line or lines inform you that the telephone directory is unchanged by the operation *QueryPersonNumber*?
14. Which variable(s) remain unchanged if a phone is disconnected?
15. Which variables are modified when a person is added to the phone directory?
16. Can an office appear in the directory if it does not have any phones in it? On which line(s) of the specification did you find the answer?
17. Can an office appear in the directory if it does not have any people in it? On which line(s) of the specification did you find the answer?
18. When adding a person to the directory what two things happen if the phone number is already associated with an office? (give line references)
19. Write the additional lines that would be required if the phone directory specification were to be modified so that it stores the department that a person works in. State where these new lines would be inserted into the specification.
20. Using your answer to question 19, write the additional lines that would be needed to add an operation called '*QueryPersonDept*' that, given the name of a person, checks which department that person is in. State where these new lines would be inserted.

Appendix 5: More stringent tests of significance

Tukeys Honestly Significant Difference (HSD) = $q\sqrt{\frac{\text{Mean square error}}{n}}$ where q is the value obtained from the Studentised Range Statistics tables. In this case the value of q (3,62) is 3.4. As there are unequal numbers in each group we use $n = \frac{3}{\frac{1}{23}+\frac{1}{23}+\frac{1}{19}} = 21.5$

This leads to a value of HSD = 6.63. This is simply used as a guide to the difference in means between the groups compared . Table 4 shows the numerical differences between means.

differences between means	A	B
B	6.13	
C	6.95	0.82

Table 4 Differences of mean scores of specifications (signs ignored)

By comparison with the Tukeys statistic we can see the differences between A and C are clearly significant and A and B are very close while B and C again show no difference. In his book [Hinton 1995] stresses the need to use judgement on those differences which almost reach the level of significance. In conjunction with the results of the t-tests we can take the difference between A and B to be important.

It is perhaps surprising to see such a low figure for the comparison between B and C. The Scheffé test allows a post hoc comparison of particular aspects of the experiment while leaving others out. In this case we can form a weighted comparison between B and C but ignore A using weights of 0, -1 and 1 as coefficients in the calculation of a F statistic of comparison.

This is found to be F = 2.425 which must be compared with the table value of F(1,62) = 4.00 and so is a confirmation of all the other tests that there is no significant difference between Specifications B and C.