Interpretation of simulation for model-based design analysis of engineered systems

Jonathan Bell

Department of Computer Science
University of Wales
Aberystwyth

May 2006

This thesis is submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy of The University of Wales.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)

Date ....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date ....

Statement 2

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)

Date ....

ABSTRACT

This thesis attempts to answer the question *Can we devise a language for interpretation of behavioural simulation of engineered systems (of arbitrary complexity) in terms of the systems' purpose?* It does so by presenting a language that represents a device's function as achieving some purpose if the device is in a state that is intended to trigger the function and the function's expected effect is present. While most work in the qualitative and model-based reasoning community has been concerned with simulation, this language is presented as a basis for interpreting the results of the simulation of a system, enabling these results to be expressed in terms of the system's purpose. This, in turn, enables the automatic production of draft design analysis reports using model based analysis of the subject system.

The increasing behavioural complexity of modern systems (resulting from the increasing use of microprocessors and software) has led to a need to interpret the results of simulation in cases beyond the capabilities of earlier functional modelling languages. The present work is concerned with such cases and presents a functional modelling language that enables these complex systems to be analysed. Specifically, the language presented herein allows functional description and interpretation of the following.

- Cases where it is desired to distinguish between partial and complete failure of a function.
- Systems whose functionality depends on achieving a sequence of intermittent effects.
- Cases where a function being achieved in an untimely manner (typically late) needs to be distinguished from a function failing completely.
- Systems with functions (such as warning functions) that depend upon the state of some other system function.

This offers significant increases both in the range of systems and of design analysis tasks for which the language can be used, compared to earlier work.

ACKNOWLEDGEMENTS

Much of the research undertaken for this thesis was part of the work of the SoftFMEA project, funded by the EPSRC under the critical systems program.

Of members of the department, I would especially like to acknowledge the assistance of, and thank, my supervisor Dr. Neal Snooke and Professor Chris Price, the other investigator on the SoftFMEA project. They have provided invaluable comments on (and enthusiasm for!) the work in progress as well as proof reading this thesis.

I would also like to thank the two colleagues with whom I shared on office during this work, Stuart Lewis and Richard Shipman, for interesting and profitable discussions; and the other members of the Model Based Study Group in the department.

On a personal level, my thanks to Fran for putting up with an at times hard pressed husband, my parents for everything and the Aberystwyth Friends (Quakers) for helping me keep my balance.

Thank you all.

To Fran, my parents and everyone else who brought me here.

Contents

1 Introduction **11**
1.1 Motivation 12
1.2 Research questions and original work 15
1.2.1 Original contribution of the present research 16
1.3 Scope of the present research 16
1.4 Structure of the thesis 17

2 The problem area **19**
2.1 Tasks for model based reasoning 19
2.1.1 Design analysis 19
2.1.2 Support for the design process 25
2.1.3 Diagnosis 25
2.1.4 Explanation and instruction 26
2.1.5 Discussion 26
2.2 The systems to be modelled 27
2.2.1 Partially achieved functions 27
2.2.2 Intermittent and sequential behaviour 28
2.2.3 Untimely achievement of function 29
2.2.4 Dependent functions 29

3 Model based design analysis **30**
3.1 Model based reasoning for design analysis 30
3.2 The nature of models for reasoning 31
3.3 Qualitative and quantitative models 32
3.4 Simulation for design analysis 34
3.5 Ontologies for modelling 39
3.6 Interpretation and simulation 40

4 Knowledge for reasoning **45**
4.1 Classes of knowledge for model building 45
4.1.1 Structural knowledge 48
4.1.2 Behavioural knowledge 50
4.1.3 Functional knowledge 54
4.1.4 Teleological knowledge 63
4.1.5 Using the four classes of knowledge 66
4.2 Relationships between the classes of knowledge 72
4.2.1 Relations between models within the grid 76
4.2.2 Discussion of the model sets 85

5 Representation of function 87
5.1 Requirements of a functional representation 88
5.2 Logic and functional description 91
5.3 The trigger 96
5.4 The effect 98
5.5 Representation of purpose 99
5.6 Relationship with the simulation 101
5.7 Functional description and report generation 107
5.8 Notation used 112
5.9 Discussion 113

6 Functional decomposition 118
6.1 Subsidiary functions 120
6.2 Using subsidiary functions 123
6.2.1 Subsidiary functions with AND and OR 124
6.2.2 Subsidiary functions and exclusive OR 132
6.2.3 Functional decomposition and unexpected effects 133
6.3 A logical basis for subsidiary functions 135
6.3.1 Subsidiary functions and AND 138
6.3.2 Subsidiary functions and OR 139
6.3.3 Subsidiary functions and XOR 141
6.4 Relationships between system and function 142

7 Incomplete representations of function 146
7.1 Purposive incomplete function 147
7.2 Operational incomplete function 154
7.3 Triggered incomplete function 157
7.4 Incomplete functions in functional decomposition 161
7.4.1 Asymmetrical functional decompositions 163
7.5 Differences between behaviour and purpose 164

8 Describing functions that depend on terminating, intermittent and sequential behaviour 167
8.1 The strict sequence operator 171
8.1.1 The sequence operator and temporal orderings 176
8.1.2 Using logical NOT 177
8.2 The loose sequence operator 178
8.3 Describing functions that depend on cyclical behaviour 182
8.4 Temporal aspects of a function's trigger 185
8.5 Using the sequential operators 187

9 Timing and function 192
9.1 Representation of temporal constraints 193
9.1.1 Consequences of untimely achievement of function 202
9.2 Temporal constraints using duration of intervals 202
9.3 Possible representations of timings 205
9.3.1 Qualitative modelling of time 206
9.4 Timing constraints in use 207

10 Dependencies between system functions **210**
10.1 Warning or telltale functions . . . 212
10.2 Fault mitigating functions . . . 217
10.3 Interlocking functions . . . 220
10.4 Recharging functions . . . 223
10.5 Relations between functions . . . 227
10.6 Simulation of dependent functionality . . . 228
10.7 Uses of modelling of dependent functions . . . 230

11 The Functional Interpretation Language in use **232**
11.1 The Functional Interpretation Language and design analysis . . . 233
11.1.1 Seat belt warning system . . . 233
11.1.2 Simulation of the seat belt warning system . . . 235
11.1.3 Functional description of the seat belt warning system . . . 240
11.1.4 Mapping between the functional and system models . . . 246
11.2 Other possible rôles for the language . . . 251
11.2.1 Diagnosis . . . 252
11.2.2 Functional specification of system design . . . 253
11.2.3 Explanation generation in education and training . . . 257
11.3 Functional description of software . . . 257
11.4 Relationship with the simulation engine . . . 261
11.4.1 Sequential behaviour and temporal constraints . . . 262
11.4.2 Simulation through the design lifecycle . . . 263
11.4.3 Dependent functions . . . 264

12 Conclusion and future work **267**
12.1 How the Functional Interpretation Language meets its requirements . . . 267
12.2 Future work . . . 272

A Formal description of the Functional Interpretation Language **275**
A.1 Elements of the language . . . 275
A.2 States of device functions . . . 276
A.3 Subsidiary functions . . . 277
A.4 Sequential operators . . . 279

B Notation used for functional description **281**
B.1 The language . . . 281
B.1.1 Labels . . . 281
B.1.2 Operators . . . 282
B.1.3 Functional states and relations . . . 282
B.1.4 Temporal constraints . . . 283
B.1.5 Other keywords . . . 283
B.2 Conventions used in this thesis . . . 284

C Common elements of diagrams **285**

D Description of functional decomposition tables **287**

Glossary **292**

References **294**

List of Figures

4.1 Grid to show possible model relationships . . . 74
4.2 Minimum set of AutoSteve models for FMEA . . . 79
4.3 Models used in the behavioural approach . . . 81
4.4 Models used in Functional Representation . . . 82
4.5 Models in CFRL . . . 83
4.6 Models used in Multimodeling . . . 84
4.7 Models for Compositional Modeling . . . 85

5.1 Circuit diagram for the torch example . . . 109
5.2 Model relationships using the proposed functional representation . . 115
5.3 The relationships between the elements in the torch functional description . . . 116

6.1 Combining effects using AND . . . 124
6.2 Combining subsidiary functions with AND . . . 126
6.3 Combining subsidiary functions with OR . . . 128
6.4 Combining effects with OR . . . 130
6.5 Model relationships for the hob example. . . . 143

7.1 Five functional decompositions . . . 162

8.1 Using logical operators resulting in a single time slot . . . 168
8.2 Finding function at the end of simulation might miss significant effects . . . 169
8.3 Specifying sequences of immediately successive effects . . . 172
8.4 Using SEQ to specify effects that should start together. . . . 173
8.5 The loose sequence operator allows less closely constrained temporal relationships . . . 179
8.6 Two temporally unrelated effects. . . . 180
8.7 Using OR to constrain unordered sequences . . . 181
8.8 A cycle of behaviour that continues until a further trigger. . . . 182
8.9 Using the sequence operator in the seat belt warning system. . . . 188
8.10 Synchronising the horn and lamp in the seat belt warning system. . 189

9.1 A simple temporal constraint on achievement of a function. . . . 194
9.2 Adding temporal constraints to a sequence of effects. . . . 196
9.3 Accumulation of tolerance if timing is taken from the initial trigger. 196
9.4 Using L-SEQ to specify timing of a sequence. . . . 198
9.5 L-SEQ does not specify duration of preceding effect. . . . 198

11.1 Schematic for the seat belt warning system. . . . 234

C.1 Common symbols in the model relationship diagrams 285
C.2 Common symbols in the function composition diagrams 286

List of Tables

5.1 Achievement of function using trigger and effect. 92
5.2 Part of an FMEA report for the torch example. 110
5.3 States of a function and the resulting text. 111

6.1 Achievement of a function in terms of subsidiary functions. 136
6.2 Functional decomposition using AND 138
6.3 Functional decomposition using OR 140
6.4 Functional decomposition using XOR 142
6.5 Functions associated with a car's exterior lights. 143

7.1 Decomposition using partial incomplete functions with AND 150
7.2 Decomposition using partial incomplete functions with OR 150
7.3 Decomposition using partial incomplete functions with XOR 152
7.4 Functional decomposition using triggered incomplete functions with AND 159
7.5 Functional decomposition using triggered incomplete functions with OR 160
7.6 Functional decomposition using triggered incomplete functions with XOR 160

8.1 Using SEQ to describe different temporal relations 176
8.2 Describing partly ordered temporal relationships 180

9.1 Part of a design analysis showing late achievement of a function. . . 208
9.2 Part of a design analysis showing late achievement of a function with no specified consequences. 209

10.1 Telltale and warning functions 212
10.2 Fault tolerant functions 217

11.1 Part of an FMEA report for the seat belt system example. 250

D.1 Example functional decomposition table 287
D.2 Example of decomposition using PIFs 289

Chapter 1

Introduction

The research described herein is concerned with the idea that automated model based design analysis of engineered systems depends on more than the ability to simulate the behaviour of the system. The ability of a simulation to derive knowledge of a system's behaviour can only describe that behaviour in terms of internal variables, such as the level of current in an electrical system. For many design analysis tasks, the finished output is a textual (or tabular) report describing the effects of unexpected aspects of the system's behaviour and the work of interpreting the (internal) results of the simulation is still left to the engineer. To automate these design analyses, what is required is a way of automating this interpretation of the results of the simulation, presenting the results in terms of the system's purpose. This requirement entails some means of mapping aspects of the system's behaviour to achievement of its purpose.

The major component of the original work presented here is a language to support this mapping of a system's behaviour to its purpose to allow this interpretive task to be included as part of an automated design analysis. This enables the automatic generation of a textual design analysis report that describes the behaviour of the system in terms of how well it fulfils its intended purpose rather than in terms of changes to internal values. The thesis is therefore based upon the idea that there are two stages to automated design analysis.

- Simulation of the system is carried out to establish its behaviour, given knowledge of the system structure and behaviour either of the components that make up the system or the physical laws underlying the system's domain(s).
- Interpretation and comparison of the results of the simulation in terms appropriate to the design analysis being undertaken, allowing useful compar-

isons to be made either between the (simulated) behaviour of the system and its intended behaviour or between the behaviour with some internal failure and its behaviour when working correctly.

Most work in the field of model based reasoning has been concerned with the simulation of systems' behaviour while the present work proposes a language to allow the automation of interpretation of the results of such a simulation in terms appropriate to the design analysis task. It does this by enabling the description or specification of the functions of a system, where function is taken to mean, in general terms, a description of how the system fulfils its purpose, mapping a system's behaviour to its purpose. This knowledge of the function of the system can be used to interpret the simulated behaviour of the system, identifying those significant changes of behaviour that should be included in an automatically generated draft design analysis report. The relationship between different classes of knowledge in model based design analysis, such as knowledge of behaviour, function and purpose, is discussed in Chapter 4.

1.1 Motivation

The modelling of engineered systems using knowledge of function (Sticklen *et al.*, 1989; Iwasaki *et al.*, 1993) has been in use for a number of years, both for deriving the behaviour of a system from knowledge of its structure and the function of its components, and for interpreting the results of a qualitative simulation (in which the system behaviour is derived from the system's structure and component behaviour and / or behavioural rules associated with the system's domain) in terms of the system's purpose. See Chapters 3 and 4 for a fuller discussion of approaches to modelling and simulation of engineered systems.

Historically, most work with functional modelling has been concerned with using knowledge of the function of a system's components to derive the behaviour of the system as a whole. This functional knowledge can be used to support various design tasks. A design can be developed by refining a functional model based on the purpose of the system until the functions can be related to individual components (Iwasaki *et al.*, 1993). The purpose of the system is the need it is intended to meet while its function or functions are concerned with how it meets that need. In this case, the system functions are decomposed in terms of connections between components. Functional modelling has also been used to support diagnosis (Sticklen *et al.*, 1989) and Failure Mode Effects Analysis (Hawkins & Woollons, 1998). In all these cases, system function is expressed in

terms of component functions which are related to each other primarily in terms of the connections between components, so capturing the structure of the system. The functional descriptions used are adequate for systems where the individual component functions are simple (such as a wire conducting) but many systems' functions depend on behaviour of greater complexity, requiring a more expressive language for their description.

Another use of functional knowledge is interpretation of the results of simulation in terms of the purpose of the system as a whole. Either a numerical or qualitative simulation tool can be used to establish the behaviour of the system being analysed. Knowledge of the system's functions maps this behaviour to the system's purpose, allowing significant changes of behaviour to be identified. This is particulary valuable for design analysis tasks such as Failure Mode Effects Analysis (FMEA) where the engineer must generate a report showing the effects of component failures on the system as a whole (Price, 2000). That report will be couched in terms relating to the intended purpose of the system, so instead of noting that a failure results in no current flowing through a car headlamp, for example, it will note that the headlamp fails to light, and the road will not be lit and the legal implications. This task is a good candidate for automation as it is extremely repetitive and it is best carried out early in the design process, so any changes found necessary can be made easily, and analysis can be repeated whenever changes are made to the design of the system, so the effects of such changes can be established. While a simulation tool will help with the analysis, the interpretation will still be the task of the engineer. Sneak Circuit Analysis (SCA), in which the system is analyzed to ensure there are no unexpected current flows resulting in unintended system outputs (Price *et al.*, 1996a; Savakoor *et al.*, 1993), is another design analysis task where similar arguments apply. The need for interpretation of results of simulation for model based design analysis is discussed at greater length in Section 3.6.

If the interpretation of the simulation is to be automated (so a draft design analysis report can be produced completely automatically) then some way of mapping the system's behaviour to its purpose is needed. One approach to this that has been found to be useful is "functional labelling" (Price, 1998). Functional labels are used to identify the system's outputs and to associate them with the purpose of the system, identifying those outputs required for the system to fulfil a given purpose. This approach has been found to work well for design analysis of many electrical systems and is in use in a commercial design analysis tool that allows the automatic generation of FMEA and SCA reports of electrical circuits in the automotive sector. It has the advantage that the functional models are simple, as

only those components whose inputs or outputs are also inputs or outputs to the system as a whole need explicit representation in the functional model. The function of other components is (implicitly) derived from their behaviour. Also, the functional model has no need of failure mode functions as the component's failure modes (that is, faulty behaviours) are associated with its reusable behavioural model. The functional models are also reusable for systems having a similar purpose, so the functional model for a car lighting system, for example, can be kept and reused for the corresponding systems in future models. All that need be changed is the mapping between the component level functions and the state of the actual components.

While this use of functional knowledge does differ from that of other workers in the field, there is common ground. The modelling of function in terms of the inputs to and outputs from a system is not inconsistent with the definition of function as "[a device's] effect on its environment" in (Chandrasekaran & Josephson, 1996). Also the mapping between purpose and behaviour and input and output in functional labelling means that the approach specifies the function as the "expected behaviour" consistently with the notion of function in (Iwasaki *et al.*, 1993). While the aim of the research is a language to support the interpretation of behavioural simulation of engineered systems, it is suggested that there is enough similarity between these notions of function that the proposed language should also be useful for increasing the expressiveness of the functional decomposition in ways that are appropriate for other functional reasoning tasks, besides the interpretive one. These other uses of the language presented as the main result of the research are discussed in Chapter 11.

With the increasing use of microprocessors in modern systems, it has been found that the functional labelling approach is limited by the expressive power of the language used to describe the system functions. The present research is concerned with the development of a more expressive language for functional description that allows functional modelling of systems whose functionality depends on behaviour of greater complexity, specifically in the following areas:-

- Systems where a function might be partially achieved.
- Systems whose functionality depends on intermittent or sequential outputs.
- Systems where there is a danger of the required outputs being achieved in an untimely manner (typically after an undue delay).
- Systems with subsidiary functions (such as monitoring or fault mitigating "back up" functions) whose achievement depends on the state of some other

system function.

These requirements are discussed in Section 2.2. In addition, to increase the range of tasks for which the new language is applicable the relationship between different aspects of the representation of function has been more closely defined, as has the nature of a hierarchical decomposition of function, following on from the work in (Snooke & Price, 1998). The primary result of this research is therefore a language for the description of function that is applicable both to a greater range of engineered systems and to a greater variety of design related tasks.

1.2 Research questions and original work

The aim of the present work is to devise an appropriate language to allow the interpretation of the results of model based simulation of a system in such a way as to enable the automatic generation of a (draft) report describing the system's behaviour in terms appropriate to the design analysis task.

The question the research attempts to answer is *can we devise a language for interpretation of behavioural simulation of engineered systems (of arbitrary complexity) in terms of the systems' purpose?* The thesis presents such a language, together with a discussion of its appropriateness both for the task of automatic generation of design analysis reports and also for other related tasks. There is also discussion of the relationship between the simulation and interpretation tasks of a model based design analysis tool. This language expresses the behaviour of the system in terms of its functionality rather than in terms of its internal state.

There has been earlier work in Aberystwyth on the use of the functional description of an engineered system for the interpretation of model based simulation, described in (Price, 1998) and (Snooke & Price, 1998). The present work builds on this earlier research by presenting a language that supersedes the earlier language by providing a more complete logical basis for description of system function and by increasing the language's expressive power. This enables the description of the functionality of systems whose behaviour is of greater complexity than could be handled by the earlier language. This earlier work resulted in the development and marketing of a commercial model based design analysis tool which, before the acquisition of the company set up to market the tool, was known as *AutoSteve*. This name has been used throughout this thesis, but it should be noted that all references to *AutoSteve* are to the tool as developed at the University of Wales Aberystwyth, not to more recent developments by any other organisation. It is

believed that all references to the tool are to materials already in the public domain. The tool, now known as Capital Analysis, is described in the web site of Mentor Graphics, the current developers[1].

1.2.1 Original contribution of the present research

The contribution of this research is a language for the description or specification of a device's function. The language supports the mapping of the device's behaviour to its purpose, allowing interpretation of the results of the simulation of a device in those terms. This allows the automatic generation of design analysis reports that state whether or not the device achieved its purpose as opposed to merely listing the values of internal variables, as output by the simulation. The language supports the description of device functions that depend on behaviour of greater complexity than earlier approaches allowed, specifically to fulfil the requirements listed in Section 1.1 above. The use of this language in design analysis is discussed, including how it contributes to the generation of design analysis reports. There is also discussion of how the language might be used in other activities, such as supporting the design process. The language is for use in a design analysis tool that combines the simulation of the subject system and its interpretation. This raises the question of what relationship exists between the simulation and interpretation tasks of such a tool. There is discussion of this relationship, specifically of the requirements placed on the output of the simulator if the expressiveness of the language is to be used to the full.

1.3 Scope of the present research

The language to be presented as the principal result of the research is intended to allow the interpretation of model based simulation of an engineered system's behaviour in such a way as to enable the automatic generation of a textual report describing the results of the design analysis. The language is not intended for use in simulation, instead of the conventional model based approach of deriving knowledge of system behaviour from knowledge of its structure and underlying behaviours. The language is useful for different design analysis tasks though it is most valuable in those tasks that require the repeated running and comparison of simulations, such as Failure Mode Effects Analysis (FMEA). As the aim of the language is the description of a system's behaviour in terms of its purpose, it is,

[1]The Mentor Graphics web page on this design analysis tool is at http://www.mentor.com/harness/analysis.html

perhaps, less appropriate for description of natural systems, such as ecological systems where the notion of purpose is not appropriate.

With the increasing use of software, a system's function might be implemented using behaviour of considerable complexity. The language is to be capable of describing systems whose correct functionality depends on behaviour of any arbitrary level of complexity. However, it should be noted that in these cases the simulation must itself be managed in such a way that this complexity of behaviour is captured. The relationship between the simulation and interpretation of such systems is discussed in Chapter 11. The function of a system is independent of any domain whose laws are used as the basis for simulation. A domain can be seen as having a self contained set of rules and be concerned with a set of internal variables, so as to allow a certain class of systems to be simulated. It is possible, given a suitable set of domain rules, to simulate systems without a complete knowledge of physics. An electrical system can be simulated using a set of rules and variables (such as current and voltage) applicable to the electrical domain, for example, but an electrical simulator is restricted by the rules it embodies to that domain. Similar functional descriptions could be applied to, for example, gas cookers or electric cookers, even though the design analysis tool might use different simulators, embodying different domain rules and variables. While examples have been taken from different domains and application areas, many of the examples are of electrical systems in the automotive sector. This is a result of the background to the research rather than any limitation of the language itself. The fact that these systems are familiar to most people means that they are well suited for use as illustrative examples in the thesis.

1.4 Structure of the thesis

In outline, the earlier chapters, Chapters 2 to 4 are background material, incorporating the literature review, the description of the original work starts in Chapter 5. Chapter 2 provides background material, including a description of the design analysis tasks the research is concerned with and the nature of the systems whose complex functionality causes difficulties for existing descriptive languages. This is followed, in Chapter 3, by a brief survey of model based reasoning and simulation of engineered systems so as to provide a context for the rôle of the present research. That chapter also discusses the interpretation of simulation in more detail. The different classes of knowledge used in model based reasoning and their relationships are discussed in Chapter 4, concentrating on the place of knowledge of function and purpose (teleology) in model based reasoning. There is

also a discussion of different researchers' approaches to the use of these classes of knowledge and the relationships between them. A proposed definition of function that is appropriate for the present work is also introduced in that chapter. Chapter 5 contains a description of the representation of function used in the present research, which forms the basis for the functional description language used for interpretation of simulation. This language is referred to as the Functional Interpretation Language to distinguish the specific language from a general reference to a language for description of function. The following chapters (Chapters 6 to 10) then extend this by describing the features of the language (and the underlying representation of function) that are used to describe more complex functionality.

The language is evaluated against a real world case study and its use for different design analysis tasks and other areas is discussed in Chapter 11. That chapter also discusses the relationship between the language and the simulation. Finally the conclusion discusses the degree to which the language meets the aim of the research and discusses future work.

Chapter 2

The problem area

This chapter describes two aspects of the problem area the Functional Interpretation Language is concerned with. First there is a description of the design analysis tasks that the Functional Interpretation Language can help automate, together with other related tasks for model based reasoning about engineered systems. This is followed by a more detailed discussion of the characteristics of the systems whose functions we want to describe. This includes introductions to some of the case study systems to be discussed later in the thesis, which are used for illustration and evaluation of the proposed Functional Interpretation Language.

2.1 Tasks for model based reasoning

Model based reasoning has been used for a variety of tasks which it seems worth briefly discussing here, so as to provide background for later discussion of the different approaches to these tasks. Here we are primarily concerned with reasoning about man made systems, specifically for design analysis, so this section will concentrate on such tasks. However it is worth noting that model based reasoning has also been applied to other tasks and also to natural as opposed to man made systems, and these areas will be briefly introduced so as to inform later discussion as to the usefulness of the present research in these areas. The usefulness of the Functional Interpretation Language for some of these tasks will be discussed in Chapter 11.

2.1.1 Design analysis

One model of the design process describes the process as an iterative cycle between synthesis of candidate designs and analysis of these designs so as to evaluate the

candidate design and to inform any further iterations of the cycle (Wood *et al.*, 2005). The synthesis of candidate designs is discussed briefly in Section 2.1.2 but the main focus of the present work is on automating the analysis of candidate designs.

Design analysis can be considered to be the checking of a system design so as to ensure that it meets the intended requirements and that any unintended behaviours (such as those resulting from component failures) have no unduly hazardous consequences. These are promising tasks for model based reasoning as they can be carried out by running a simulation of system design to establish its behaviour. The behaviour is derived from knowledge of the system's structure (possibly derived from a schematic drawn in a suitable CAD tool) and either knowledge of the behaviour of the individual components or the physical domain rules that underpin the system as a whole. The automation of these analyses allows them to be carried out early in the design process and repeated for successive refinements of the design as the design process progresses. It is also a good deal simpler than constructing a physical model of the proposed system design. The use of model based and functional reasoning for these design analyses, and different approaches, will be discussed more fully in the following chapters. There are broadly speaking two groups of design analysis task:-

Design verification tasks where the design is simulated to ensure that its behaviour matches the intended functionality of the system. Such tasks require the simulation to be compared with a representation of the intended functionality of the system. They therefore need such a representation to act as the basis for this comparison. These tasks include design verification itself and Sneak Circuit Analysis.

Failure analysis tasks where the behaviour of the system with some failure (typically a failure to one or more components) is compared to its behaviour when working correctly to identify the consequences of such component failures. Such analyses include Failure Mode Effects Analysis and Fault Tree Analysis.

The design analysis tasks listed as examples above will each be briefly described in the following sections.

All these design analysis tasks have in common that they are concerned with how well the system fulfils its intended function (rather than any more detailed aspects of its behaviour). They can perhaps usefully be grouped by the term "functional analysis". This term will be used to refer to this group of design analysis tasks here on in.

2.1.1.1 Design Verification

The purpose of this analysis task is to ensure that the design for the system is such that the system will fulfil its intended purpose. A model based approach to design verification might simulate the system's behaviour and compare the results of the simulation with a suitable description of the system's intended behaviour. The resulting report will naturally need to highlight any areas where the simulated behaviour fails to match the intended behaviour.

An approach to design verification has been proposed, (McManus *et al.*, 1999) based on the design analysis tool developed at Aberystwyth, in which an attainable envisionment of the system's behaviour is compared with a representation of the intended functionality. An attainable envisionment means that the system is simulated in every state that can be reached from a specified initial state. That paper suggested using a state chart as a means of representing the intended functionality. This thesis will discuss the use of the Functional Interpretation Language for this purpose, in Chapter 11.

A good deal of research, especially in the field of reasoning about function, has been done into automating design verification. (Chandrasekaran *et al.*, 1993) see the design process as one of verifying and refining candidate designs, using knowledge of the design's functionality to inform its physical (structural) design. This verification and top-down refinement of a system's functional design is also used by (Iwasaki *et al.*, 1993).

2.1.1.2 Sneak Circuit Analysis

Sneak circuit analysis ensures that there are no unexpected paths (for electric current or hydraulic pressure) through the system, resulting in unexpected (and possibly damaging) actions taking place. Where related subsystems have several switches (possibly including safety override switches) it is possible that certain combinations of switch positions will result in an electrical path being completed for some unexpected part of the system (typically with current flowing in the opposite direction to that intended). Such "sneak circuits" are often found when subsystems are combined. It can be carried out by running repeated simulations of the circuit in different configurations, such as combinations of switch positions. These simulations could be qualitative or quantitative. While running repeated quantitative simulation will reveal sneak circuits, the output of the simulator will need scanning to identify the variable values that indicate the sneak. The use of an interpretive language can automate this identification of the significant values

generated by the simulation. This technique is a good candidate for qualitative simulation, as sneak circuits can be detected from schematics of the system and are also best found early in the design process, when changes to the circuit are easily made. As the analysis requires repeated simulations (and their interpretation) it is also time consuming if done manually, so automation of the task leads to a worthwhile saving of an engineer's time.

Sneak circuit analysis appears to be an analysis that can readily be added to a tool intended for FMEA, see (Price *et al.*, 1996a), suggesting SCA as a second rôle for an automatic FMEA tool, and (Savakoor *et al.*, 1993). The latter paper discusses the combination of sneak circuit analysis and FMEA. That paper includes an example sneak circuit with potentially serious consequences, the inadvertent lowering of an aircraft's landing gear.

The name suggests this analysis is specific to certain domains, such as electrical and hydraulic, unlike FMEA. However, one of the central ideas behind SCA is that new failures might arise from the combination of subsystems and this possibility is not limited to any domain, of course.

2.1.1.3 Failure Mode Effects Analysis

It is suggested by (Bowles & Wan, 2001) that Failure Mode Effects Analysis (FMEA) is one of the most beneficial tasks in a well structured reliability program. In FMEA, the effects of known failure modes (faulty behaviours) of components or subsystems are projected to the next layer in the component / subsystem / system hierarchy. For example, an electric wire has the failure mode "open circuit" (that is, the wire breaks). Clearly its effect is that it fails to connect the components at its ends, but the effect on the system will, of course, be determined by its position in the system. The rôle of FMEA is to find the effects of all such component (or subsystem) failures. The result of conducting an FMEA is a report describing the (system level) effects of component failures, together with figures indicating the failure's severity, likelihood of detection and the probability of its occurrence. The report might also allow the action to be taken by the design team to remove the failure, and the person responsible for taking these actions, to be added. The model based FMEA tool developed in earlier work at Aberystwyth automates the generation of this report by simulating the behaviour of the circuit, both when functioning correctly and when components have failed. The automatically generated FMEA report can then be edited by hand, allowing the resulting work to be specified and allocated.

FMEA is a strong candidate for automation as it is a laborious task and also

requires sufficient knowledge of the system and the domain to allow the effects of faults to be traced. It therefore needs to be undertaken by an experienced engineer. The laboriousness of the task means that FMEA will often only be undertaken once, late in the design process, to verify correctness of the design. Automating the process allows FMEA to be carried out more readily, so it can be done more often, to see whether changes to the design are necessary, and to trace the effects of changes to the design, rather than simply carrying out one FMEA late in the design life-cycle, to confirm the safety of the system. If FMEA can be done early in the design process, any drawbacks with the design are identified in time for changes to be made in a cost effective manner. The benefit of running FMEA early makes its automation a good candidate for the use of a qualitative reasoning technique, as the necessary simulation of the system can be done with incomplete knowledge. This is valuable in the automotive industry, for example, as schematics of the electrical systems will typically be drawn before the actual components to be used are specified and well before the actual wiring runs (which depend on the design of the vehicle's bodywork) are known. Therefore, complete knowledge of the system, sufficient for a full mathematical simulation will only come late in the design process.

In many application areas, it is becoming increasingly the case that some of the intended behaviour of devices is implemented using programmable components, so software is taking a greater rôle. The increasing use of software in modern systems, has led a need for software FMEA of embedded systems, as identified by (Goddard, 2000). That paper suggests two complementary methods, system level FMEA, based on the top level design and detailed software FMEA, based on source code. This might raise interesting questions for the automation of FMEA of systems including significant software, as either model might be incorporated into an FMEA of the system in which the software is embedded. A technique for semi automatic safety analysis (including FMEA) is proposed by (Papadopoulos *et al.*, 2001).

2.1.1.4 Fault Tree Analysis

Fault Tree Analysis has much in common with FMEA, see (Price *et al.*, 1997). A fault tree relates a fault (i.e. a system failure) to its possible causes (i.e. component failure). As FMEA relates component failures to system failures, a simple fault tree can be generated from an FMEA report, by grouping the various component failures covered in the FMEA by the system failure they lead to. Clearly for the fault tree to be complete, the FMEA must also be complete, and it is also likely that there will be some component failures that must be listed in several places

in an FTA report, as the same component failure might result in several distinct system failures. (Price *et al.*, 1996c) discusses the idea of generating a fault tree from an FMEA. An FTA report can be used as a starting point in diagnosis, as the symptom can be looked up and the possible causes found.

2.1.1.5 Prioritising failures

As noted in the description of FMEA above, the seriousness of a failure might be estimated by the design engineer and this estimate expressed using three numbers between one and ten. These are multiplied together to arrive at the failure's Risk Priority Number (RPN). Any alteration made to the system as a result of the design analysis might depend on how high a failure's RPN is. Failures with a high RPN will be given higher priority for corrective work on the system. The RPN is the product of three values, each of which can take a value between 1 and 10. These are

Severity Indicates the seriousness of the consequences of the failure. A value of 1 is insignificant, 10 means the failure is likely to result in injury, death or severe financial loss.

Detection Indicates the likelihood the failure will be noticed by an operator before the failure results in its likely consequences. A value of 1 means the failure is easily detected, 10 that it is likely to remain undetected.

Occurrence An indication of the likelihood of the failure occurring. A value of 1 is highly unlikely, 10 is probable. Note that this is a property of the reliability of the component, in terms of the failure that leads to a system failure, rather than of the system failure itself. The same system failure might have different causes, each with their own value for occurrence, so a headlamp might not light because the bulb filament has blown (likely) of because a connecting wire has broken (less likely).

Note that the values for detection and occurrence are not probabilities. Indeed the value for detection varies inversely to the probability of detection. A value of zero cannot be used for any of these, of course, as the product (the RPN) will then be zero and information about the other two factors of the RPN is lost. It is worth repeating that these values are estimates and it is possible for a failure to be given different (inconsistent) values for severity and detection in different parts of an FMEA report although this should not happen. This description is useful because the values for severity and detection will be used to illustrate features of

the decomposition of system functions in Chapter 6. As occurrence is associated with the likelihood of the cause of a system failure occurring, it has less to do with the functional model of a system than with the behaviour of the component whose failure causes the fault, so is of less interest in the present work.

2.1.2 Support for the design process

There has been a good deal of research on automated support for the synthesis aspect of the design process, some approaches to which are discussed briefly in Chapter 4. One common model of this aspect of the design process is to refine the design's function, decomposing the device function into contributing functions until these functions map onto component behaviours, so guiding the selection of components and also the (schematic) structural design of the device. While the approaches differ in detail, as do the representations used, the approaches of (Umeda & Tomiyama, 1993; van Wie *et al.*, 2005) as well as (Iwasaki *et al.*, 1993) all use this decomposition of function and mapping to behaviour as a way of modelling functional design and an approach to developing computerised support for the process.

There has been work attempting to use a functional representation language to capture the functional decomposition of an intended system and use this to guide the design of the system's physical structure. For example, (Chandrasekaran *et al.*, 1993) seek to use their functional representation to capture the functional aspects of the design rationale of a product, using it to help the design process, which they see as being one of iterative refinement of the functional design of the product, evaluating alternative candidate designs.

It has been argued, by (Gero, 1990) that a product's functional design (concerned with what a product is to do) is derived from a causal mechanism, this being an intermediate stage between the requirements specification and the actual physical design. A functional modelling language has been used, by (Iwasaki *et al.*, 1993), to capture this causal structure of a proposed system.

This raises the possibility of a model based reasoning tool being used during the design process itself, rather than to verify the results of a (provisionally) completed design, as is the case with the design analyses introduced earlier.

2.1.3 Diagnosis

Diagnosis can be considered a distinct field from the analyses introduced above, as all the analyses above take as a starting point the use of a full set of known com-

ponent failures, and they reason from the component failure to a system failure. Reasoning for diagnosis, of course, works in the opposite direction as the system failure is known and we wish to find candidate component failures. A fault tree is one possible means of finding such candidates, of course. (Price *et al.*, 1996b) discusses the use of fault trees (generated from FMEA) for diagnosis.

Other research has looked at qualitative reasoning as an approach to automated diagnosis, (David & Krivine, 1986) suggests using knowledge of structure and behaviour as the starting point for system diagnosis. However, (Price & Hunt, 1989) argues that additional knowledge is needed besides the qualitative model. It is necessary as a qualitative model might lead to ambiguous results; additional knowledge, such as knowledge of the diagnosis task might be used to remove these ambiguities. That paper discusses how such additional knowledge is used by other workers in the field. Qualitative reasoning and its rôle in design analysis is discussed in the next chapter.

2.1.4 Explanation and instruction

One other area in which model based simulation and analysis can be used is in the field of demonstrating and explaining a system's working. This might be done to help with instruction of operators of a hazardous system, instructing them on the system's failures and how to react to them. The advantages of this with respect to systems whose failures are hazardous need no explanation. For example, (Tuttle & Wu, 2001) discusses the use of *CyclePad*, a thermodynamic modelling program, in instruction in thermodynamics for engineering students.

2.1.5 Discussion

The design analysis tasks introduced above are all candidates for automation using model based reasoning, especially qualitative reasoning, as the aim is to identify significant differences between the intended behaviour and the actual (simulated) behaviour, in the case of the design verification tasks and between the faulty behaviour and the correct behaviour (that is with no component failures) in the failure analysis tasks. In many cases the aim is to find system behaviours with safety implications, such as the loss of a required output (car lights failing to light) or internal behaviour changes with safety related consequences, such as electrical short circuits. In other words, these analyses are less concerned with whether there are minor changes in the operating state of the system (such as small changes in

current) than they are with changes to system state or behaviour that result in significant changes to the system's effect on its environment.

As has been noted, these analyses are best carried out early in the design process when any changes shown to be necessary are easily made and can usefully be carried out before factors that affect a full numerical model of the system (such as exact component specifications, cable runs) are known. The use of qualitative reasoning techniques allows the automation of these analyses at an early stage in the design process. However, for full automation of these analyses, more is needed than the ability to run qualitative simulations of the systems. Some way of presenting the results of these simulations in terms of the system's effect on its environment is required. This is where the ability to interpret the results of simulation is important. There is a fuller discussion of the use of model based reasoning in design analysis in Chapter 3.

2.2 The systems to be modelled

Another area introduced in Chapter 1 that might usefully be expanded on early in the thesis is some description of the characteristics of the systems that a model based design analysis tool might be called upon to model, specifically those features of these systems that cause problems for existing functional description languages. Where appropriate, this discussion will be illustrated using real-world example systems. The discussion is confined to characteristics of the systems themselves, the problems caused by these characteristics is left to discussion of the uses of functional knowledge and the languages themselves, in Chapter 4. This section is subdivided into sections dealing with each significant characteristic in turn. These subsections correspond to later chapters describing the proposed language's approach to describing these characteristics.

2.2.1 Partially achieved functions

Where a function has more than one effect, it might be necessary to distinguish between cases where the achievement of some of the effects, but not all, is better than nothing, so the failure of the function is mitigated and those where the absence of any required effect is tantamount to complete failure to achieve the expected function. For example, where a warning system has both an audible and visual warning (a telltale lamp and a warning horn, perhaps), the failure of either one of these outputs does mean that some warning is still given, so while the

warning function is clearly not correctly achieved this is arguably a less complete failure than is the case if both outputs fail. This can be contrasted with, say the failure of either car headlamp, which, partly because of the legal implications can reasonably be considered to amount to complete failure of the headlamp system, as it renders the car unusable. This area is discussed in more detail, together with an approach to representing these cases, in Chapter 6.

2.2.2 Intermittent and sequential behaviour

One aspect of many examples of engineered systems that causes problems for simple functional languages is the idea that a system function can depend on a sequence of distinct behaviours (or subsidiary functions). A simple example is a washing machine whose overall function can be decomposed into a sequence of wash, rinse and spin dry functions, each of which run to completion during the cycle and which must occur in the right order. This function cannot readily be modelled in terms of a single goal state as the goal state (as far as the machine itself is concerned) is identical to the start state — the machine is idle with clothes inside. The only difference is likely to be that the control systems (whether electromechanical switchgear or microprocessors) will be in a different state but this is not appropriate for checking the function as this only relates indirectly to the purpose of the machine. To illustrate, the switchgear might be in the required final state but if the water flow was malfunctioning, the clothes will not be clean.

In many application domains (such as the automotive sector) this characteristic results from the increasing use of microprocessors. Their use either allows existing, behaviourally simple systems to be used for more complex behaviours or the addition of new, behaviourally complex systems. An example of the first case is the use of counted flashes of a car's direction indicators to confirm remote locking or unlocking of the vehicle, for example two flashes to confirm locking, one to confirm unlocking. The second case is illustrated by the fitting of a warning system to indicate that the driver's or front seat passenger's seat belts need fastening. An example of such a system not only lights a telltale lamp on the dashboard but also sounds a buzzer intermittently for a programmed period of time. This means that once the function has been correctly achieved, the buzzer is once again idle, of course. How the proposed functional language models such functions is described and discussed in Chapter 8.

2.2.3 Untimely achievement of function

This characteristic is related to the above. What is meant in this context is that the expected effects of the function are achieved but not at the right time, relative to the action that triggers the function. A simple example might be an excessive delay between a driver pressing the dip switch and the headlamps dipping. Notice that this differs from the case where the dipped headlights are on even though they should not be. This particular example is perhaps unlikely, but cases where the lamp switching is carried out using a network using a carrier sense multiple access collision detection protocol (such as the widely used CANbus) could result in delays caused by network loading resulting in collisions delaying transmission of a message.

It will be appreciated that where a system function depends on intermittent outputs, the timing of these outputs might be significant. In the seat belt warning system mentioned above, the individual buzzes must each last a suitable length of time, so the description of the buzzing sequence might be qualified by how long each buzz and pause should be. The proposed approach to modelling cases where a function is achieved late (or early) is discussed in Chapter 9.

2.2.4 Dependent functions

The term "dependent function" is used to refer to a system function that comes into operation in response to the correct achievement (or otherwise) of some other system function. Typical examples might be a monitoring or warning function whose output is triggered by the failure of some expected system function or a fault mitigation function that is triggered by a failure in the main function. The nature of these functions is considered in more detail in Chapter 10.

One other similar area is where the action of an input depends on the current state of a system. A simple example is a toggle switch. For example, in some cars switching between dipped and main beam is accomplished by the identical action of pulling and releasing the dip switch. In this case some way of identifying the fact that a given action (operating the switch) has different consequences is needed. These consequences depend on the current state of the system.

Having introduced various aspects of the problem area, it is time to consider approaches to the use of model based and qualitative reasoning in the field of design analysis.

Chapter 3

Model based design analysis

Having introduced the design analysis tasks for which model based and qualitative reasoning offers a useful approach, this chapter introduces model based reasoning, discusses its rôles in design analysis and the approaches to model based design analysis taken by different researchers. Most work in the field has been concerned with simulation of systems, to gain knowledge of their behaviour, so this area is discussed here, although the main thrust of the present work is concerned with the interpretation of the results of simulation. This chapter will conclude with a discussion of the place of interpretation of the results of this simulation in design analysis, and discuss the relations between this need for interpretation and different approaches to simulation. This material will lead on to a discussion of the different classes of knowledge used in model based reasoning and the relationships between them, which forms the content of the following chapter.

3.1 Model based reasoning for design analysis

The field of model based reasoning has grown from that of qualitative reasoning which is an area of artificial intelligence with the aim of imitating a human expert's ability to reason with imprecise data. For the design analysis tasks outlined in Section 2.1 the qualitative approach has several advantages. The ability to reason with limited and imprecise data allows the analysis to be carried out early in the design process, before all the information necessary for a full mathematical analysis of the system is known. In addition, these analyses are concerned with analysing the behaviour of a system in terms of its functionality (how well it fulfils its intended purpose) which generally means that major changes to the behaviour are what are of interest. For example, in a power windows system, we are more interested in the fact that both windows open and close, rather than

the more detailed case that the voltage across both motors is close enough that they open or close at much the same speed. Even though this is of interest, it is not significant for FMEA or SCA, for example. Naturally the use of qualitative models of a system will lead to an analysis that captures changes to the system that lead to such qualitative (with the term used in its more colloquial sense) changes in behaviour. For example, a typical qualitative electrical model of an electrical system might use three levels of resistance (zero, load and infinite) and these will be associated with three levels current; short, active and none. Therefore any change in the system's structure must entail a substantial change in current (from active to none, perhaps, if the resistance of part of the circuit changes from load to infinite) which will clearly imply a significant change in the system's behaviour. This is not to say that there is not a rôle for numerical analysis in these functional design analyses, merely to suggest that qualitative reasoning is a useful approach to such analysis. There is more on the relationship between qualitative and numerical simulation in Section 3.3 and on how they affect the need for interpretation of the results in Section 3.6.

3.2 The nature of models for reasoning

Reasoning, in this context, is taken to mean the generation and use of knowledge not explicitly included in the original description. For example, the aim of qualitative simulation, the establishment of a system's behaviour from knowledge of its structure, is the generation of new knowledge about the system. The nature of the knowledge to be generated and the use to which it will be put will, of course, depend on the reasoning task. It will typically be the case that for a design analysis task, the required new knowledge will indeed be knowledge of the behaviour of the whole system. How this knowledge is generated will in turn depend on what knowledge is available to the reasoning system and what methods are available to work on this available knowledge. For tasks dealing with failures of the subject system, such as FMEA for example, we might generate a model of the behaviour of the correctly working system and a model of the behaviour of the system incorporating the failure and then establish the effects of the failure by comparing the two models. For design verification the generated model of the system's behaviour might be compared with some suitable description of the system's required behaviour.

The idea of generating new knowledge from the input models suggests that they need be executable, though (Leitch *et al.*, 1999) notes that the model itself need not be executable, if the reasoner includes some sort of simulation engine to work

with the model. In an electrical design analysis tool, for example, the structural model of the circuit need not itself be executable, if a circuit analysis engine (such as "CIRQ" (Lee & Ormsby, 1991; Lee, 1999a)) provides the necessary facility to allow reasoning (in this case the derivation of the system's behaviour) to take place using that model. In contrast, a state chart might be considered an executable model of a system. It can certainly be suggested that some way of working with the model is needed.

Another important feature of the models used as the basis for reasoning is that they must capture sufficient knowledge of the system to be analysed. There are two aspects to this. The model must capture sufficient knowledge of the system to be a sound basis for the reasoning process and the system itself needs to be sufficiently self contained to allow all the necessary knowledge to be represented. If a system's behaviour is to be derived, we need to be able to model all the (significant) influences on the system. In other words we need to be able to define the boundaries of the system such that the closed world assumption holds true, and the boundaries must themselves be such that the amount of knowledge they must contain does not lead to a model of intractable complexity. This is one reason why electrical systems have been commonly used as subjects for qualitative reasoning. The domain knowledge is relatively simple and electrical systems are sufficiently self contained that the closed world assumption holds in that inputs to and outputs from the system can easily be defined and other effects on the system (such as heating from the environment) can be ignored without so simplifying the models that useful reasoning can no longer be done.

A related aspect of whether a model is adequate for the analysis is the possibility that for the design analyses discussed in Section 2.1 that are concerned with analysing the system in terms of its intended functionality a detailed simulation might require the use of more information than is necessary for the analysis concerned, leading to excessive overhead in capturing the information. This case, especially as it relates to the relationship between numerical and qualitative simulation is discussed in the next section.

3.3 Qualitative and quantitative models

Model-based reasoning is concerned with building tools capable of reasoning about physical systems from first principles, rather than by attempting to explain the system in terms of rules specific to that system, as is the case with first generation expert systems. In contrast the rules governing a model based system are more

general in scope, such as qualitative versions of the physical laws that underpin a system's behaviour. It has grown from the field of qualitative reasoning (QR), to include quantitative and mixed models as well as those seeking to capture qualitative behaviour. This, of course, introduces an immediate distinction between using conventional mathematical models of a system, such as underpin the numerical electrical circuit analysis tools Saber (Saber, 1996) and SPICE (Quarles *et al.*, 1980), and qualitative models.

It might be supposed that if mathematical (numerical) models are available, there is no reason to use a less precise alternative. However, there do appear to be good reasons for using qualitative techniques alongside conventional numerical analyses. This section will discuss these and the differences of approach between using qualitative and numerical analyses. The electrical domain provides a good example of this, as there exist both numerical and qualitative tools for circuit analysis.

Conventionally, an engineer will seek to use a mathematical model of the system under consideration. Tools exist to allow computerised analysis of electrical circuits using mathematical models, notably Saber and SPICE. These tools allow a variety of sophisticated analyses. These analyses are detailed and are concerned as much with how well a system works, how efficient it is for example, as with how it works, such as whether it fulfils its purpose. It is quite possible to argue that there is a place for more high level analyses besides these. It will be noticed, for example, that this list does not include either FMEA or SCA. This is not to say that such tools cannot be used to help with such functional analyses of electrical systems, but there are difficulties with so doing.

One difficulty with using the sort of numerical analysis undertaken by these tools is that it needs considerable interpretation. A change in the system might alter the potential across some significant component. The analysis will produce a listing of all changes so will include small changes of no significance as well as any substantial changes. It is left to the user to interpret this list, to decide which changes are significant. Some design analysis tasks, including FMEA and SCA, are concerned with such significant changes to the circuit's operation. A qualitative analysis of the system will naturally highlight significant changes, such as those to the system's topology, but will not capture relatively small parametric changes. If the results are to be interpreted in terms of system function, then this is more easily done from a qualitative analysis because of this highlighting of major changes to the system. This is valuable for design analysis tasks such as FMEA. How an interpretive language relates to this difficulty is introduced later.

If a numerical analysis of the system is to be done, full mathematical models of all

the components are needed. One aim of qualitative reasoning was to imitate the human reasoner's ability to "reason with less data and less precise data" (Forbus, 1988) than is necessary for a full mathematical analysis. The availability of all the data necessary for mathematical modelling is especially unlikely early in the design life cycle of a system, when functional refinement of a design might result in its schematic being drawn before the actual components are specified. Indeed, in the automotive sector, a full numerical analysis will need to wait until such details as the cable runs are known, which information depends on the design of the vehicle's bodywork and so will not be available until late in the design process.

These difficulties mean that there is a rôle for a simpler qualitative tool, especially for use early in the design process. Qualitative reasoning is an established branch of artificial intelligence that seeks to capture the intuitive knowledge of engineers and scientists and which they will use to guide their analyses of the problem. As electrical systems are typically self contained with a well known set of physical laws, that domain has been an important one for research into qualitative reasoning and various approaches have been tried.

3.4 Simulation for design analysis

There has been a variety of approaches to qualitative reasoning about systems and as an aim of the present research is to establish relationships between these approaches, and between the different models used, much of the rest of this chapter and most of the next are concerned with discussions of these different approaches.

Arguably the most fundamental distinction between approaches is concerned with the model of behaviour from which the system behaviour can be derived. One approach makes use of some qualitative physics, such as that proposed in (Hayes, 1985) or failing that a set of domain specific laws to provide a global view of a system's behaviour. In practice, a more realistic target seems to have been to develop a useful set of rules sufficient for a specific domain (such as modelling electrical circuits). The series - parallel - star reduction approach to reasoning about electrical circuits of (Mauss & Neumann, 1996) is an example, discussed shortly. The alternative approach reasons from knowledge of individual components' behaviour or function. The functional reasoning approaches of (Sticklen *et al.*, 1989; Sembugamoorthy & Chandrasekaran, 1986) and others are examples of this approach. These contrasting approaches to reasoning about the behaviour of a system are discussed in more detail in Section 4.1.2.

These qualitative physics based approaches contrast with the causal reasoning

approach of (de Kleer, 1984). That paper uses causal reasoning to derive the behaviour of a circuit from its structure, and then uses teleological reasoning to determine the circuit's function. He is interested in reasoning about how a circuit reacts to perturbations in its inputs. The circuit is modelled as a connected graph of component models, each of which interacts with its neighbours, those components with which it is connected. A component model is used when there is an input to that component (such as a change in voltage or current). Components only interact with their neighbours, so preserving a link between the behaviour and structure of the circuit and only act directionally, so there is no negotiation between components - a component's input cannot be changed. This generates a causal explanation of the circuit's behaviour. This, of course, relies on the notion that a change causes other changes, when actually they occur simultaneously, but is a useful model of an electrical engineer's intuition. The other area of this paper, the use of teleological reasoning to link behaviour and purpose, is briefly discussed in the section on knowledge for reasoning, section 4.1.4. This differs from the qualitative physics approaches in not being dependent on domain laws.

One possible difficulty with this approach is scalability as a complete causal net may be of intractable complexity. One approach to this problem is to make use of knowledge of a device's function in order to organise causal knowledge of the device. This is one of the goals of Functional Modelling (Sticklen *et al.*, 1991). Knowledge of device function is used to inform the construction of and so reduce the complexity of the causal net. This maintains the domain independence of a causal (as opposed to a qualitative physics based) simulation while reducing the problem of scalability. There is more on the uses of functional knowledge in Section 4.1.3.

One difficulty with any causal reasoning based approach is that it cannot be used to find the behaviour of systems that cannot be reduced to a chain of causality. Examples of such systems include electrical circuits where it cannot be shown which components are active, as directions of current flows cannot be found. It is a weakness of techniques based on local information (such as component models) that they miss global information that might add useful extra knowledge about the circuit. This is a motivation for techniques that reason about a system's global structure. Examples of this approach can be found in the field of circuit analysis.

One classification of approaches to qualitative modelling of systems that relates closely to the contrasting approaches noted above is to distinguish between those that reason about interactions of components with their neighbours, such as (de Kleer, 1984) and those that reason about the global structure of the system, such as (Lee & Ormsby, 1991; Mauss & Neumann, 1996; Milde *et al.*, 1999) in the

electrical domain. By "global" in this context is meant knowledge of the structure of the system as a whole.

These "global" techniques can treat the circuit as a graph and replace constraint propagation with graph analysis. Mauss and Neumann (Mauss & Neumann, 1996) use series-parallel-star (SPS) reduction to reduce a circuit to a single resistance. This resistive edge has a known direction of current and voltage drop, as it joins power and ground. The act of reducing the circuit generates a tree which is then used to trace the flow and voltage drop back to the original edges in the circuit graph. This reduction allows circuits incorporating bridges to be modelled, but the simple three valued qualitative resistance model will generally not allow the direction of flow through a bridge to be established, it must be labelled as ambiguous. The SPS reduction can be used in combination with a quantitative resistance model, so directions of flow are found through bridges.

A similar approach is used by Milde at al (Milde *et al.*, 1999) as they also reduce the circuit to generate a tree to represent the circuit structure. They use qualitative analysis of the network to generate a diagnostic decision tree for the circuit. They differ from both (Mauss & Neumann, 1996) and (Lee & Ormsby, 1991) in using deviations from the reference values for resistance, voltage and current, as in diagnosis it was felt important to model the effects of slight parameter deviations. Components have expressions to describe their behaviour, following FLAME (Pugh & Snooke, 1996). In this approach, a component's behaviour is modelled as an expression relating its input and output. For example, a relay might be modelled as "if there is current through the coil, resistance across the switch is low". A component behaviour is instantiated, the qualitative currents are found and if the instantiated behaviour results in a violation of an internal condition of the model, another component behaviour is used. Clearly if the all component models result in such a violation of the internal condition, behaviour prediction fails. This technique was intended for diagnosis, and as such can assume that the original, correctly functioning circuit's behaviour can be predicted. This might not be the case if the topology gives rise to ambiguous current flows. The paper does not suggest how this difficulty is approached.

A different simplification of the circuit is used by CIRQ (Lee & Ormsby, 1991; Lee, 1999a). The circuit is represented as a graph of nodes, representing terminals, and edges representing connections, with a value for resistance. This algorithm also uses three values for resistance (zero, load and infinite) and the circuit is simplified by combining the nodes connected by zero resistance edges into so-called "supernodes". The simplified graph is then traversed, finding the distance of each node from power and ground. These values represent the qualitative potential

at each node, and can be used to derive the current flow, as current flows from points of higher potential to points of lower potential. This allows directions of current to be found for a circuit with no bridges but the same difficulty arises with bridges as with (Mauss & Neumann, 1996), the three values for resistance are not sufficient to find unambiguous flows. Clearly, where the original circuit has bridges some of whose surrounding edges are of zero resistance, these edges are removed in deriving the supernodes, so the circuit will no longer contain the bridge. This algorithm is simpler than Mauss and Neumann's SPS algorithm, but gives identical results in three-valued qualitative analysis.

This problem with finding direction of flows through bridges has led to some work in the use of more than three values for resistance. A development of CIRQ has been devised and implemented, (Lee, 1999b; Lee *et al.*, 2001). This allows any number of values for resistance, provided they are of different orders of magnitude. The idea of an order of magnitude follows (Raiman, 1986) and means that there are no cases where a series of resistances add up to a total resistance as great as the next higher level, so each level is insignificant relative to the next higher level. This uses series parallel reduction to simplify the circuit and a variation of CIRQ's finding of potentials at the nodes in the circuit graph to find directions of current. This allows bridges that are unbalanced by an order of magnitude to be identified and the unambiguous current flow to be found.

AutoSteve (Price, 1998) uses CIRQ as a global (domain level) simulator combined with component models that incorporate behaviour in a similar way to that described in the discussion of the technique of (Milde *et al.*, 1999). There is more on the relationship between the structural and behavioural aspects of *AutoSteve* in Chapter 4, but it is worth noting here that the component modelling has been extended by allowing the use of state charts (Harel, 1987) to model components with internal memory, as described in (Snooke, 1999).

In these cases, (Pugh & Snooke, 1996; Milde *et al.*, 1999; Snooke, 1999), the behaviour of a component is specified by the user rather than being derived from a complete qualitative physics (Hayes, 1985; Forbus, 1984), as was proposed as a target for qualitative reasoning. This makes the theoretical foundation less complete than was intended by Hayes, as the bottom level of the system decomposition is expressed in terms of knowledge of how components work rather than by physical laws. However, it does capture the knowledge of the engineer who is designing the system being analysed about electrical components in easily built models, rather than relying on a library of domain theoretical models describing the behaviour of components. Although a complete qualitative physics might have laws that allow the behaviour of a relay, for example, to be derived, this will be a good deal

more complex than a simple description of its behaviour. In principle, a qualitative physics could, perhaps also derive failures of a component as is needed for FMEA, but modelling the component in such detail as to allow, say, the ingress of dirt to cause a relay to stick open seems impossible, as too much will need to be known about the physical construction and environment of the component. It seems a good deal simpler to allow the tool to capture the engineer's knowledge of component's failure modes. It is worth noting that both *AutoSteve* and Milde et al's tool are intended for use by engineers, and have been developed as working tools, so their usability is important. In practice, of course, component models can generally be retrieved from a library. Simplicity of modelling and re-use of models is important if a tool is to be used, there is little gained by automating a task if the automation results in (or is felt to result in) more work than carrying out the task manually.

All of these global reasoning techniques have in common the fact that they depend on the circuit being represented by a network of resistances and that the analysis is of the circuit in a steady state. This limits the components that can readily be modelled — capacitors, for example, whose behaviour is dynamic, cannot be included. In some application areas this is not too problematic, as such components are rare. *AutoSteve*, for example, is capable of analysing the great majority of circuits in the automotive sector. However, there are limitations on the use of qualitative analysis of electrical circuits, both in the circuits that can be analysed and the kinds of analysis. As both FMEA and SCA depend on the topology of the circuit (the kind of steady state DC analysis of which qualitative reasoning is capable) qualitative simulation is a suitable technique for these tasks. This does, however suggest that there are complementary rôles for qualitative and quantitative analysis.

Indeed *AutoSteve* has been extended to make use of a quantitative circuit analysis tool (Saber). This means that later in the design life cycle, once components have been specified, a more detailed analysis can be carried out for FMEA, allowing possibly ambiguous current flows through bridges to be determined as well as the level of current across fuses, so showing whether the fuse will blow correctly.

If a global view of the system is to be used, as in the cases discussed above, then this needs to be combined with a global view of behaviour, as can readily be done in the electrical domain. It could well be argued that it is impossible to derive a system's behaviour purely from its structure, there must be some suitable notion of behaviour either locally (as in the constraint propagation approach) or globally (as in a qualitative physics) from which the system behaviour can be derived. The quoted aim of qualitative simulation being "to derive behaviour from

structure" is something of an over-simplification. Other domains might either have a less convenient set of possible domain laws to capture this global behaviour, or indeed none at all. One example of a domain where domain rules are not appropriate is modelling of software. In this case a global view of the proposed system architecture will have different rôles, such as tracing the dependencies of specific variables through the system. There is more on the relationship between global and local models of behaviour in section 4.1.2.

3.5 Ontologies for modelling

In the context of model based reasoning, by ontology is meant the nature of the basis for modelling the system that is to be analysed, of what the model is primarily composed. It can be seen as defining the chosen view of the nature of the system being modelled. The model of the system might be composed principally in terms of it components, the processes it embodies or the constraints that act upon the possible values of the system's variables. All of these alternative views might rely on the same set of variables, so a model of an electrical system will use variables for voltage, current and resistance, for example, whatever the ontology chosen.

The component centred ontology, see (de Kleer, 1984), describes a system primarily in terms of its components. So a locomotive boiler, for example, will be viewed as an assembly of a fire box and its water jacket, a pressure vessel, several fire tubes and an inlet for the water (with a feed pump) and outlet for the steam, and so on. This is a natural ontology for the design analysis of engineered system, of course, as they are built up from components and this view of the system will be readily available for analysis, as in the way an electrical simulator uses the schematic as the basis for its model building.

The process centred ontology takes the alternative approach of viewing a system as a collection of related processes, see (Forbus, 1984). Instead of the physical components, the boiler will be viewed in terms of the processes of flow of water in, steam out and the heating of the water by the fire. This ontology has been used for modelling engineered systems, and also has a close relationship with the idea of functional decomposition for design. For example, a domestic washing machine's wash function can be decomposed into the filling, heating, agitation and draining processes. The idea of refining the functional specification of a system as part of the design process is discussed in (Iwasaki *et al.*, 1993). In this approach to design the functional and physical designs are refined in parallel, one might use knowledge of the functional design to inform selection of components. In the washing machine

example, the breaking down of the wash function leads to identification of a need for a connection to a water supply, further decomposition of the requirements of the fill subsidiary function will reveal the need for valves and filters in the inlet system.

The constraint centred ontology views the system in terms of the constraints on variables within the system. It is introduced in (Kuipers, 1986). Of the three ontologies it is perhaps closest to conventional mathematical modelling. For example, in modelling a boiler a constraint centred model will capture the relationship between the pumping of water into the boiler and the boiler pressure — if water is being injected, the pressure will tend to reduce. It is arguable that these models are more abstract than is helpful for many design analysis tasks, a simpler view of the system will be sufficient.

For the design analyses introduced in Section 2.1, the component centred ontology might well be seen as the most natural and most work in the area of design analysis does use this ontology. In failure analysis (such as FMEA), especially, what we are concerned with is tracing the effects of the failure of a component on the system and its environment. The fact that component failure modes are known (and reusable) pieces of behavioural knowledge at a component level, seems to support this view. The functional refinement of a design proposed as a model of the process by (Iwasaki *et al.*, 1993) in which function is refined until a function can be related to a specific component can be seen as relating a process centred view of the system to a component centred one.

3.6 Interpretation and simulation

While qualitative and model based reasoning have long appeared to be a useful approach to automating design analysis (and other tasks), the approach has led to the introduction of very few commercially useful tools. One problem with the approach, at least from this point of view, could be argued to be the concentration on the simulation of the subject systems, as this is insufficient to automate the whole task of design analysis. It is suggested in (Price *et al.*, 1997) that one of the challenges an automated design analysis tool must meet if it is to be adopted is that it should offer support for the whole process. To illustrate this idea that a tool should support the whole design analysis process, let us consider the process of conducting FMEA using a mock up of the system and using a model based tool that simulates the behaviour of the system.

FMEA is conventionally undertaken by assembling the system, using examples of

the actual components so that its behaviour can be explored and the failure modes introduced enabling the resulting behaviour to be compared with the behaviour of the system working correctly. It will be appreciated that even a simple system will have a good number of components, each with its failure modes whose effects need to be assessed so this is a time consuming and repetitive task. Firstly, the system's behaviour when working correctly must be established, by assembling the system and noting what happens in response to the various inputs (for example when throwing switches). Then each failure mode is introduced in turn and the same programme of alterations to the system inputs is repeated and changes to the behaviour of the system (especially its outputs) relative to the correct behaviour are noted and a report describing these effects is compiled. A model based (or mathematical) simulation of the system can be expected to save time and might be more reliable, if derived from the design schematic, as the possibility of the inadvertent incorrect assembly of the test system is eliminated. If a simulation tool is available, then the system can be simulated working correctly and the behaviour noted as before and the failure mode behaviours can readily be modelled as these changes can be inserted into the simulation tool. The drawback of this approach is that the effects on the outputs of the system still need to be assessed by the engineer. For example, suppose a numerical electrical simulation tool (such as SPICE) is used, then a comparison between the correct and failure mode simulations might show that there is a different level of current in parts of the circuit. It is left to the engineer himself to assess how these changes affect how well the system fulfils its intended purpose, by translating these internal changes to the system behaviour to the state of its outputs. It could be argued that the use of a simulator on its own is therefore not an unmitigated improvement. It saves having to set up the test system and having to physically model the component faults but against this, the mock up of the physical system will show the effects on the significant system outputs. For example, any lamps will be seen not to come on. As the results of the simulations need assessing manually, then the design analysis tool is limited to running one simulation at a time and presenting the results to the user of the system (the engineer) for assessment and comparison with the intended behaviour (if the analysis is concerned with design verification) or the correct simulation (for failure analysis). Therefore the use of a simulation or modelling tool does little to alleviate the repetitive nature of such design analyses as FMEA or SCA. Indeed as it will involve the interpretation of a complete set of simulations of the whole system it might even lead to more work, as a human engineer will have enough knowledge to avoid mentally simulating parts of a system that will be unaffected by a certain failure.

Therefore if the advantages of model based simulation of systems are to be fully realised for design analysis it is necessary to find some way of enabling the design analysis tool to present the results of the simulations in terms linked to the purpose of the system. If this is done, the output of the tool can highlight the important features of the behaviour of the system. For example, the simulator might show that there is no current through the one side of a headlamps circuit by listing all components in which the level of current has changed, while once the purpose is known, all that need be shown is that there is no current through one of the headlamps themselves. Also, where the analysis is numerical, the tool might be able to distinguish between changes of system behaviour that result in a small change to the level of current (for example) in some part of the system and behaviours where the change in current results in some important change in system output. A simulator with no capability for interpretation of its results will inevitably list all changes of system state or behaviour now matter how insignificant. Therefore, one important rôle for interpretation of simulation is to provide a useful basis for the comparison between different simulations in failure analyses such as FMEA, so that these comparisons can also be handled automatically. A related rôle is to provide a similar basis for comparison between the intended and (simulated) actual behaviour of the system in design verification.

Previous work in Aberystwyth has used the approach of "functional labelling" (Price, 1998) in which significant system states (or, usually, outputs) are associated with intended system functions. This is discussed in more detail in Section 4.1.3. This approach has been shown to be successful in allowing the automatic generation of design analysis (specifically FMEA and SCA) reports from model based simulation and an important motivation for the present work is to increase the expressiveness of the functional description language used to allow the approach to be used for a greater variety of engineered systems.

Different simulation techniques might be used as the system's design lifecycle proceeds, allowing different analysis tasks to be done as more detailed information about the system becomes available. Design analysis might begin by using a simple three valued qualitative model, progressing through a more sophisticated qualitative model (such as an order of magnitude based one) to a numerical model, as the design process continues, as proposed in (Price *et al.*, 2003). Running successively finer-grained simulations allows a greater variety of analyses to be carried out and also allows more detailed results of repeats of earlier analyses to be obtained. This might serve to disambiguate earlier results, for example by showing whether or not a fuse will blow, preventing an otherwise uncontrolled short circuit. For this to be done and textual design analysis reports to be generated

automatically the language used for interpretation of the results must be capable of being mapped either to qualitative or numerical simulations. The commercial tool developed from the earlier work at Aberystwyth does this, to allow either three valued qualitative or numerical simulations (using Saber) to be compared using the functional labelling approach to interpretation. A refinement of the idea of comparison between the results of different simulations is the notion of an incremental FMEA, (Price, 1996). In that paper it was suggested that the consistency of automatically generated FMEA reports enabled two such FMEA reports of different versions of the same system to be compared by the automated FMEA tool and a report to be generated that showed only the changes from the earlier report. This means that FMEA can be carried out early because it is quick and cheap to re-do the FMEA in response to changes to the system design. The use of an interpretive language for this comparison allows the generation of a similar incremental FMEA report following a finer grained simulation of the system, comparing its results with an earlier coarse grained simulation. These differences have to be identified using the interpretive language as the two simulations concerned might have results that are incompatible — the current flow through some significant component might be "medium" in a qualitative simulation and 2 Amps in a later numerical simulation. The respective mappings between the simulations and the interpretive (functional) description of the system provide the means to establish whether these two values can be regarded as equivalent, in so far as they lead to the same system behaviour or state. The idea of such incremental design analysis reports comparing analyses at different granularities is discussed in (Price *et al.*, 2003) and a similar incremental approach to generating diagnostics is discussed in (Price, 2002).

It was suggested in Section 3.2 that for simulation the system needs to be modelled such that the closed world assumption applies. However, for automatic generation of design analysis reports, some knowledge of the systems environment is required. For example, an FMEA report of a car headlamps circuit might say that the left headlamp fails to light in full beam so the road ahead is not properly lit and further note the legal implications. An ideal reasoner might know all this, of course, but as we have seen, all a model based reasoner's simulation is likely to reveal is the loss of current through the part of the system that includes the left hand headlamp's main beam filament. The interpretation of the simulation can and should allow the capture and use of knowledge other than that required for the simulation itself in presentation of the output of the design analysis tool. Since the report will typically be concerned with unexpected effects of the system on its environment or on unexpected loss of intended effects, some way of weakening the

closed world assumption is required for interpretation of results. This is discussed in more detail in the next chapter.

As the rôle of interpretation is concerned with the results of the simulation, it need not matter how the simulation is carried out. However one possible use of an interpretive language that might be of interest is in providing a basis for comparison of a functional decomposition of a newly designed system, as proposed by (Iwasaki *et al.*, 1993) with the resulting component based behavioural model of the same system, as an aid to verifying the correctness of the design.

This chapter has suggested that an aim of model based simulation is to derive knowledge of the behaviour of a system from knowledge of its structure and that the functional design analyses are concerned with how well the system fulfils its intended purpose. Therefore we are concerned with different classes of knowledge, both of the system itself and (some aspects of) its environment. The following chapter considers these classes of knowledge in more detail and explores the relationships between the commonly accepted classes of knowledge used in model based reasoning.

Chapter 4

Knowledge for reasoning

As suggested at the close of the previous chapter, there are various classes of knowledge held to be useful for model based reasoning. At the simplest, for model based simulation there is a requirement for knowledge of the subject system's structure and some representation of underlying behaviour, whether derived from domain rules associated with a qualitative physics or representations of the behaviour of components at the bottom level of decomposition of the system.

The following section discusses these classes of knowledge. It incorporates separate subsections for each of the four classes of knowledge commonly held to be of interest in model based design analysis. These will introduce different researchers' uses of these classes of knowledge. This section will conclude with a discussion of the definitions of each class of knowledge used throughout the remainder of the thesis. There follows a discussion of the different relationships between these classes of knowledge used by different researchers.

4.1 Classes of knowledge for model building

There are frequently taken to be four classes of knowledge used in model based reasoning. These were summarised as follows in (Chittaro & Kumar, 1998).

Structural knowledge describes system topology. The main question it answers is "Which components are in the system?".

Behavioural knowledge describes the potential behaviours of components. The main question it answers is "How do components work?".

Functional knowledge describes the rôles components play in a system. The main question it answers is "What do components do?".

Teleological knowledge describes the purpose assigned to a system by its designer or users. The main question it answers is "Why is a component in the system?".

While there is a fair degree of agreement about the usefulness of these four classes of knowledge there is less agreement about their uses. Indeed it is frequently the case that a researcher's approach will not actually use all four classes of knowledge. The nature of structural knowledge is generally agreed, perhaps not surprisingly as it is (generally) distinguished by its static nature as opposed to the more dynamic nature of the other classes of knowledge. By this is meant that the components of which a system will not change during simulation, assuming a component centred ontology. The static nature of the presence of components in the system contrasts with the accent on what components do and why in the summaries of the other three classes on knowledge in the list above. In particular, different researchers have quite different ideas of the rôle of functional knowledge. Some researchers, especially in the functional reasoning community use it in simulation and so have an idea of function that is closer to behaviour than those researchers who use models of behaviour and have a notion of function that might be very close to that of purpose. These areas will be considered in more detail in the separate subsections as will the different approaches to these classes of knowledge.

These classes of knowledge are related. Each class can be regarded as an abstraction of the class below, provided there is no interest in how the item modelled achieves its results. Naïvely, we can say that if a system is fulfilling its purpose, we can say that its function is achieved, if its function is achieved it is behaving correctly and if it is behaving correctly we need take no interest in its internal structure. These abstractions are, of course, not helpful when the purpose is not being correctly fulfilled as we are likely to want to know why not and what behaviour might be occurring instead.

As a more concrete example of the idea that behaviour abstracts structure, an electrical relay might have four terminals, two for the coil and two for the switch. If current flows through the coil, the magnetic field generated will cause a core to move, so changing the position of the switch, and allowing current to flow between that pair of terminals. A structural description, combined with knowledge of electromagnetism reveals this. However, it might be that for the purposes of the analysis being undertaken, we can simplify the modelling by ignoring the structure and simply using a behavioural model of the relay. This model might capture no more knowledge than the idea that if current flows between one pair of terminals (those associated with the coil), then resistance between the other pair of terminals is changed to zero or, if we prefer to remove all reference to structure, that flow

of current is enabled (though not guaranteed) between the switch terminals. We cannot state that there is a flow between the switch terminals as that implies knowledge of the present behaviour of the reset of the system. This does suggest that we can say that behaviour abstracts structure. The relationship between the other classes of knowledge is more complex in that they depend on the context of the system or component, what Chandrasekaran (Chandrasekaran & Josephson, 1996) calls the "mode of deployment". This is because a component might have several possible functions, and so several functional models, not all of which will be used in a given system. For example, a wire might transmit electricity, transmit a physical force, if pulled, or generate a sound if plucked. Which of these modes of deployment is used will depend on the system.

The fact that different researchers have used these terms in different ways and the resulting overlap between them, especially between behaviour and function and between function and teleology had already been noted by (Franke, 1993). Franke proposed a set of definitions not dissimilar to those listed above that stressed the contextual differences between the classes of knowledge. His definition of structure has it describing the entities (either components or processes) that make up the system. Behaviour is defined in that paper as the description of the structure ontology, within and at the boundary of the structural entity (such as the component), so that the behavioural model is self contained, rather than being defined in terms of the system. This definition does appear to differ from that offered by (Chittaro & Kumar, 1998) in so far as there is no idea of explanation of how the component works. Function is defined as the descriptive elements of the structure ontology, expressed in terms of the behaviour of the system. This does give a quite clear distinction between the idea of behaviour and function. Finally teleology is defined as the elements of the structure ontology considered in terms of the requirements or specifications of the system. This does lead to a distinct differentiation between the classes of knowledge, but these definitions do not appear to have resulted in the hoped for standardisation of usage of the terms.

The remainder of this section will consist of a consideration of different researchers' definitions and uses of each of these classes of knowledge in turn, before a discussion of these classes of knowledge (most especially functional knowledge as being most relevant to the present work) in which definitions used throughout the remainder of the thesis will be introduced.

4.1.1 Structural knowledge

It is suggested that if the behaviour of a system is to be derived either from knowledge of the behaviour of the individual components or from some global knowledge of the behaviour of the system's domain(s), then the reasoner at least needs knowledge of what component is connected to what, so it can trace the paths of changes in state of components through the system. There is a fair degree of agreement between workers on the nature of structural knowledge. The definition in (Chittaro & Kumar, 1998) has already been quoted, while (Kaindl, 1993) has "a structural description deals with components and their relationships". The idea that structure can be represented in terms of entities and their connections seems common, though whether these entities are components depends on the ontology, so (Franke, 1993) notes that the entities might be (physical) components or processes. Another possibility is that in (Keuneke, 1991), where components are related according to their functional, rather than topological, relationships. Keuneke notes that this might result in a different structural diagram. She uses as an example the difference between a topological deconstruction of a telephone, which might be split into chassis and handset, and a functional one that might relate components concerned with transmission of signals as opposed to those concerned with their reception. Each of these functional groupings will, of course, include components in both the chassis and the handset. It might be argued that a process based structural description will be not dissimilar to a functional one.

In Section 3.4, the idea that there are two contrasting approaches to the structural model was discussed. These are those that have a global view of the structure, such as is needed by an electrical circuit simulator, see (Mauss & Neumann, 1996; Lee & Ormsby, 1991) and those that simply add knowledge of what a component is connected to (and how) to the component models, so there is no global view. The causal approach of (de Kleer, 1984) and the functional approach of (Sticklen *et al.*, 1989) are examples. Possible difficulties with this latter approach to representing structural knowledge (or strictly approaches to simulation associated with it) were raised in Section 3.4. It is, of course, the case that the structural model used by the reasoning engine need not be the same one that a user sees. It would be quite simple to derive knowledge of what each individual component is connected to from a schematic (which encapsulates a global view of the system) or indeed to derive a global view of the system from a complete set of component models' connections, so at the structural level, both approaches can be seen as equivalent.

One important difference between these approaches, if a component centred on-

tology is used, is that the global view of a system must include knowledge of the components' internal structure if it is to be analysed using a domain level simulator, such as CIRQ. Both (Mauss & Neumann, 1996; Lee & Ormsby, 1991) model an electrical circuit as a network of resistances. This network of resistances has to be constructed both from the schematic (if that is how the circuit's connections are captured) and from the internal structure of the components. This combination of a global circuit model with components structural models is described in (Price *et al.*, 1997). The resulting design analysis tool provides a user interface to allow these component structural models (which are reusable) to be drawn. This need for knowledge of a component's internal structure could make modelling of components difficult in cases where the system designer has insufficient knowledge. An example might be a system that interacts electrically with a complex Electronic Control Unit (ECU), as the ECU will typically be supplied by an external contractor, and will be a "black box" as far as the system designer is concerned. If no global view is used, we need no knowledge of a component's internal structure. This provides a neat definition of what is meant by a component in terms of the decomposition of the system, as it will be modelled purely in terms of function or behaviour, rather than structure in that there is no need for a description of the internal structure of the component. In other words, the idea of a component is placed unambiguously at the bottom of the hierarchy of elements of the system, rather than being underpinned by a model of domain level behaviour.

The difference between these contrasting approaches is illustrated by the relay example above. Where there is a global view of the system combined with a domain based simulator, the effect of a component's behaviour might be described in terms of change to the internal structure of the component. In the case of the relay, "when current flows through the coil, resistance between the switch terminal becomes zero, else it is infinite". If there is no domain level simulator, that description is not helpful, and will be replaced with one along the lines of "if there is current flowing between the coil terminals then flow is enabled between the switch terminals".

Structural knowledge is commonly held to differ from the other classes of knowledge by its static nature, in that the components in the system do not change during the simulation. There are two areas where the static nature of structural knowledge does not apply. One is where the connections between components change during simulation. The relay is a good example of this as during an *AutoSteve* simulation the circuit will be simulated (using CIRQ) and if there is found to be current in the relay's coil then the switch will close, changing the resistance of the switch and so adding an electrical connection between the components pre-

viously separated by the open switch. This does, of course, change the structure of the circuit. The other area in which the static nature of structural knowledge does not apply is where a process based ontology is used. While new components are not added to the system during simulation it can certainly be the case that running a simulation will result in individual processes starting or stopping.

4.1.2 Behavioural knowledge

It has already been suggested that the primary aim of model based reasoning (certainly when used for design analysis) is to generate knowledge of the behaviour of the system as a whole. This aim was often stated as "deriving behaviour from structure" but it is the case that some knowledge of underlying behaviour (either at a domain or component level) is also required. The early aim was to establish a qualitative physics (Hayes, 1985) that consisted of underlying knowledge of qualitative versions of physical laws that would (in principle) allow the behaviour of any system to be derived from knowledge of its structure. It might, for example, be feasible to derive a model of the behaviour of an electric motor from such a physics and then incorporate this behavioural model (or these behavioural models) into model based simulation of a larger system. This approach does have drawbacks, however. The first is the amount of work required in generating the necessary underlying physics, and the difficulty of devising suitable qualitative versions of physical laws. Related to this issue is the fact that many qualitative models of behaviour at a component level are very simple. To return to the motor example, it is hardly worth qualitatively simulating a motor (even if the physics is available) to learn, essentially, that if you put a flow of current through the windings, you get torque at the shaft. This simple model of behaviour is something that engineers know, so is readily captured by the system without such low level simulation. If there is no underlying set of laws from which we can derive the behaviour of any element, then this suggests the need for some approach to capturing knowledge of behaviour at the bottom level of the component / subsystem / system hierarchy.

Different researchers are concerned with behaviour at different levels of the component — system hierarchy. As noted in the introduction to this section, (Chittaro & Kumar, 1998) defines behavioural knowledge as describing the potential behaviours of components. That is, they are concerned with the use of behavioural knowledge as an aspect of the basis for simulation. In contrast, (Kaindl, 1993) has a behavioural description representing the potential behaviours of a system. This might be read as suggesting that this sees the behavioural description as the target of simulation rather than as knowledge to be used in reaching the target. Kaindl

is concerned to distinguish between behaviour and function and sees behaviour as describing how a function is achieved. (Keuneke, 1991) is also concerned with this distinction and also has the rôle of behaviour as explaining how an expected result (function) is achieved. The overlap and distinction between function and behaviour will be considered in greater detail in the following subsection.

One aspect that these various definitions of behaviour have in common is that they are concerned with representing the state of the system or component itself, so there is no explicit representation of the system's (or component's) environment. Function, by contrast, typically does place the effect of a device's behaviour in the context of its surroundings, as will be discussed in the next section. This local nature of behaviour is illustrated by the definition in (Franke, 1993) where the behavioural representation consists of the descriptive elements of the structure ontology (that is components or processes) within and at the boundary of the structural entity. An apparent example of this distinction between behaviour and function is (Sasajima *et al.*, 1995) which has behavioural representation simply as "necessary and sufficient information for simulating state changes". In other words it has nothing about what the state is intended to achieve. Instead it is concerned solely with the system in isolation. The other definitions cited here also represent behaviour in terms of state, though (Franke, 1993) has a sequence of states where states map variables onto values as an example of a behavioural representation, as opposed to (an element of) a definition.

As suggested earlier, there are two approaches to behavioural modelling. The original intention of the qualitative reasoning community, of developing a qualitative physics that would result in a set of physical laws available for deriving the behaviour of any system, is typically simplified to a set of domain laws, allowing systems in that domain (such as electrical systems) to be analysed. For example the approaches to electrical analysis in (Lee & Ormsby, 1991; Mauss & Neumann, 1996; Milde *et al.*, 1999) all model the circuit as a resistive net, allowing the use of relatively simple domain rules, sufficient to model and simulate most (but not all) circuits. Exceptions include circuits that include capacitors if the domain model has no rule for modelling charge. An alternative approach is to model behaviour as local to a component and trace the influence of that behaviour through the system.

In practice a combination of these approaches might be used. Both (Price *et al.*, 1997; Milde *et al.*, 1999) use a domain based simulator to model the behaviour of the circuit as a whole, combined with component behavioural models that will affect the structure of the circuit and so cause changes to the circuit's behaviour. These researchers use different approaches to modelling the circuit, (Milde *et al.*,

1999) uses the series — parallel — star approach of (Mauss & Neumann, 1996) while (Price *et al.*, 1997) uses CIRQ, (Lee & Ormsby, 1991). However they both use similar component models that have dependency expressions to model changes to the component's internal structure as a result of its behaviour. The relay example used earlier might have the dependency expression

$$\text{if coil.}i = \text{ACTIVE then switch.}r = \text{ZERO else switch.}r = \text{INFINITE;}$$

where i is (qualitative) current and r is resistance. The meaning is, of course, that resistance between the switch terminals is zero if and only if there is current in the relay's coil, otherwise it is infinite. The use of these behavioural models means simulation is iterative. (Price *et al.*, 1997) describes how the domain level simulator (CIRQ, in this case) will be run on a circuit in its initial state and then the components' behavioural models are checked to ascertain if any changes to the circuit topology result from the state of the circuit established by the simulation. These changes are made and the process repeated until a steady state is reached. For example, if the initial run of CIRQ shows that there is current flowing through the coil of a relay, the resistance of its switch is changed to zero and CIRQ run again with the changed circuit. The software component that manages this interaction between the circuit's structural model and the components' behavioural models is described in (Price *et al.*, 1997).

This use of a library of component behaviour models avoids the overhead of creating a sufficient set of domain rules to allow components such as (in the electrical domain) relays and motors to have their (typically) simple behaviours to be established by additional simulation. These component models are reusable as they are independent of the context of the component. It will be appreciated that this is consistent with the idea that behaviour is local. This does require some care in modelling component behaviour. An electric motor will generate torque if there is current in the windings, but whether this torque results in a rotation will depend on the state of the system, for example that all bearings are correctly aligned and lubricated so as to allow rotation of the mechanical components to which the motor is connected.

Where this approach is used, some components need no explicit representation of behaviour. A wire (used as a connector in an electrical circuit) has no need for such an expression as its behaviour will not change during simulation and any changes to the wire (such as failures) can be represented by changes to the structural model. For example, if a wire is to be modelled as broken (for failure analysis), its resistance can simply be changed from zero (assuming a qualitative model) to infinite. If it is to be modelled as creating a short circuit than that

can be modelled by adding a zero resistance connection between the wire and the component it is shorted to.

In contrast, it might be the case that some components have an internal behaviour that is too complex for representation by a simple dependency expression. The increasing use of electronic control units (ECUs) in electrical systems is a case in point. In these cases a more complex model of behaviour might be required. For example, (Snooke, 1999) describes the use of state charts to model complex component behaviour. The system adopted for the tool as developed actually differs slightly from the description therein, as a component's interaction with the system is through its electrical terminals rather than the ports discussed in the paper. The ports were intended to allow non electrical interactions.

State charts are an extension of state transition diagrams for finite state machines, see (Harel, 1987). They add three features:

- Hierarchical decomposition. A state can have subsidiary states so that to be in the state is to be in one or other of its subsidiary states, with transitions between the subsidiary states and also transitions between the so-called "super state" and other high level states. A transition leaving the super state entails leaving whichever subsidiary state is active.
- Concurrency. A super state can be divided such that to be in the super state is to be in more than one of its subsidiary states concurrently. There will typically be two (or more) concurrent subsidiary state charts, both of which are active when the super state is active. Leaving the super state entails leaving each concurrent group of sub states.
- Communication. A state chart might fire a transition which has an associated action firing a transition in another part of the state chart, or in the case of many state chart based languages, another related state chart.

As such they are a useful formalism for representing components such as ECUs whose behaviour implements logic rather than being modelled by domain rules. Strictly speaking, of course, such components could conceivably be modelled in terms of domain laws with a model of the governing logic (the software) being used to guide the iteration through the domain level simulations. This seems unduly complicated, however, as this would reduce the simulation to operating at the gate level of a logic component when all that is needed is an adequate model of the governing logic. Naturally, if what is modelled is the logic, this depends for correctness on both the model used for simulation being an accurate model of

the actual component's implementation of the intended logic and that implementation being itself being correct. A possible approach to modelling software for design analysis, specifically FMEA, is proposed in (Snooke, 2004). The approach allows dependencies between variables to be traced statically, showing the effects of erroneous values. The intention is that either the source code or a graphical model of the code can be used for such an analysis so this would help ensure the correctness of the models used in design analysis by avoiding the need to create a further model of the software (such as a state chart).

The division between a domain based simulator for simulating the system as a whole (or at least for the whole of that part of a system associated with a specific domain) and component behavioural models, as in (Price *et al.*, 1997; Milde *et al.*, 1999) works best if all interactions between components are simulated at the domain level, so each component model is self contained. Modern systems, however, in many application areas are making increasing use of data buses for passing signals between parts of the system. This means that some interactions within a system are best modelled at a behavioural level in much the same way as complex component behaviours and for the same reason, that there is no benefit in modelling such digital signals at the bit level, even assuming an electrical simulator is available. This raises the difficulty that the correct correspondence between the behavioural and structural model is less well maintained. An electrical circuit simulator can readily use the system structural model (possibly derived from an ECAD schematic) as its primary source but if message passing is modelled at the component behavioural level, using state charts, then the structural model must be checked specifically to ensure that there is a suitable connection between the transmitting and receiving components. Note that all a component behavioural model can do is make a signal available to the network. It cannot model its transmission as that depends on the system outside the transmitting component. This loss of the simulation's reliance on the structural model might suggest an increased rôle for functional knowledge in these cases.

4.1.3 Functional knowledge

Of the four classes of knowledge discussed here there is arguably least agreement about the nature of functional knowledge. There is a good deal of difference between how different researchers treat this class of knowledge. While (Chittaro & Kumar, 1998) defines functional knowledge in terms of the rôles components play in the system, answering the question "what does a component do?", that paper notes that function is actually an overloaded term. Besides this so-called

"purposive" definition that is concerned with the relation between behaviour and purpose (teleology), that paper offers an alternative definition of function. The alternative, "operational", definition defines function as a relation between input and output of energy, information or material. It is perhaps closer to the mathematical idea of a function. In addition to the two definitions, (Chittaro & Kumar, 1998) also gives two alternative formalisms for representing function.

Associational defines the function of a component in terms of its effect on the system or environment. This might relate a component's function to a system purpose, so there is an association between this representation of function and the purposive definition above. It works well when there is a one to one mapping between component and output.

Definitional defines function in terms of low level primitives, such as flow and state. In state based models, the semantics of each unit of function is defined by the model builder. This introduces the problem that different model builders will define function differently. In flow based models, a function is defined as a relation between input and output, which suggests an association with the operational definition of function. A set of functions is defined for the modelling system, not by the model builder. Definition of a complete set is a difficulty, but given a sufficient set, it is less subjective than the state based approach.

These representations differ in that the associational representation associates a function with the world outside the system, with the purpose or effect of achieving the function, while the definitional representation associates a function with values of flow or state that are intrinsic to the system model itself. This distinction supports the idea that the associational representation is readily associated with the purposive definition of function, while the definitional representation can be associated with the operational definition.

Function is a relational concept in that it establishes relations between a system's (or component's) behaviour and its purpose. This can be thought of as abstracting behaviour as, if we are to assign a function to a component, then in analysis of the system, we need not be concerned with how that component fulfils its function, merely that it does. It can be argued that function in this view simplifies the effects of a behavioural representation, as the system's state (resulting from the behavioural description) is represented in terms of some goal state, as in (Sasajima *et al.*, 1995). While this abstraction might be useful if a device is fulfilling its function it is less useful if the device is failing to do so, as we are likely to want to know more about the effect of the faulty device than that it is not fulfilling its

function. This is, therefore, arguably not a useful abstraction for failure analysis, as we wish to model what happens when a component fails to fulfil its function, and in doing so model the effects of faulty behaviour. The behaviour of some component that is failing to fill its intended function might have other, unexpected, side effects on the behaviour of the system.

Different workers frequently have their own definition of function, but these can in general be placed within the context of the overall framework outlined above. Simplistically there are two main schools, one sees function in a more or less behavioural context, contrasting it with the idea of purpose while the other sees function in terms of purpose. An extreme case of this latter approach is that of (Kaindl, 1993) which simply has function as the intended purpose of a component, so that the function of a steam release valve is to prevent a boiler explosion. An example of the opposite, behavioural, approach is that of (Navinchandra & Sycara, 1989) where function is represented in terms of inputs and outputs. Function has also simply been defined as a device's effect on the flow of material, energy or information through the device, without reference to the physical processes that cause the effect (van Wie *et al.*, 2005). This can readily be seen as an abstraction of behaviour. The absence of reference to physical processes also seems consistent with the idea in Functional Modeling (Sticklen *et al.*, 1991) that functional decomposition of a system can be stopped at a level where "world knowledge" can be treated as the cause of some function's effect. This allows the function of a motor, for example, to be explained simply in terms of what motors do. This is therefore not dissimilar to the idea of a dependency based (or even state based) model of a component's behaviour, in that that model also abstracts away the underlying domain rules.

One area where there is some common ground between these two schools is the idea that function is to be considered in the context of the enclosing system or its environment. While (Kitamura *et al.*, 2002) notes there is no consensus on definition of functionality of artifacts, that paper suggests that most functional models represent intended goals or rôles of the behaviour and then they are dependent on the context of the components in contrast with behavioural models [that are] independent of context. This agrees with the definition of function in (Franke, 1993) as the descriptive elements of the structure ontology considered in the context of the larger, enclosing mechanism or system. This idea of function as being in the context of the component's (or system's) surroundings seems to raise interesting questions about the nature of component function and its use in simulation, which will be discussed in Section 4.1.5. The idea that function is less local to a device than its behaviour is suggested by (Sembugamoorthy & Chandrasekaran, 1986)

defining function as what the response of a device is to some external stimulus, as opposed to behaviour explaining how this result is achieved and also by the definition of function in (Chandrasekaran & Josephson, 1996) as a device's effect on its environment. That paper gives three desiderata for a representation of function. It should:-

- Apply to intended functions of human designed devices and to functions or rôles in natural systems.
- Apply to functions of both static and dynamic objects (e.g. support).
- Apply to functions of both abstract and physical objects (e.g. software modules or steps in plans as well as physical systems).

There is a formal definition of function as effect in (Chandrasekaran & Josephson, 1996):

> Function: Let G be a formula defined over properties of interest in an environment E. Let us consider the environment plus an object O. If O (by virtue of certain of its properties) causes G to be true in E we say that O performs, has or achieves the function (or rôle) G.

This definition fulfils the three desiderata. It does not have any specific notion of purpose, which strengthens its applicability for describing natural systems, where the idea of purpose or design might be philosophically debatable.

While there is general agreement that function and behaviour are distinct concepts, (Loganantharaja, 1993) notes that there is some confusion and that it is sometimes assumed that function and behaviour are the same thing. It is, however, arguable that this confusion is less common than failing to distinguish function and teleology (purpose). Most researchers' approaches to and definitions of function have some idea of purpose, intention or goal, so that (Kampis, 1987) suggests that the concept of function enters into a system when some purpose is supposed to b accomplished by the system and (Chittaro *et al.*, 1993) defines function as the relation between a device's behaviour and the goals assigned to it by the designer. A related idea is that function is an interpretation of behaviour (Wood *et al.*, 2005). That paper notes that function is more than a subset of behaviour and implies a notion of intent on the part of the designer, following (Chandrasekaran & Josephson, 2000).

The field of functional reasoning seeks to make use of functional knowledge to assist in reasoning about a device's behaviour and functionality. This arises from the

idea in (Sticklen *et al.*, 1989) that knowledge of the function (purpose) of a device enables an organisation of our causal knowledge of the device. It is suggested by (Sticklen *et al.*, 1989) that reasoning about function is useful with reference to the three crux issues identified in (Davis & Hamscher, 1988) as important for model based reasoning. These are domain independence, scalability and model selection. It is claimed that the functional reasoning approach is intermediate between the deep reasoning approach, which entails the production of a complete causal net for the system (leading to problems with scaling) and naive physics that is based on a set of domain rules. Representing components in terms of their function avoids the need both for the causal net and the domain rules, so the approach addresses the first two of the three issues. This leads to the idea of Functional Modeling (Sticklen *et al.*, 1991) whose aim is to use the known functionality of a device to organise causal understanding of the device (so avoiding the need for a complete causal net) and provide a reasoning algorithm that can simulate the device for given starting conditions. There might be felt to be a possible question about the soundness of this approach (at least for some design analyses) as if the organisation of knowledge is used to avoid a complete causal net there does seem to be a danger that possible unexpected behaviours will be missed by the analysis. The importance of capturing the unintended but inevitable functions (by which is presumably meant behaviours) in FMEA is noted in (Wirth & O'Rorke, 1993).

One use of function in simulation is demonstrated in (Hawkins & Woollons, 1998), who use a flow based operational definition of function for FMEA. Each component has a functional rôle model, such as 'generator' or 'conduit'. The components' rôle models are combined in functional process models that show how components combine in processes. These processes, in turn, combine to create phenomena, and these phenomena relate to system goals, the teleological model. The paper only deals with the rôle and teleological models. The teleological model represents the designer's goal for the component, such as "transfer", "keep" etc. The paper links these goals to activities in the rôle model. The idea of defining functions for components can be argued to have the benefit of domain independence, enabling FMEA of complex, mixed domain systems. The paper example uses a system with an electrically driven rotary pump. The approach has been demonstrated, but it does seem to suffer from the difficulty associated with flow based representation of function, in that there is a need for a full set of functional primitives. An additional difficulty is arguably the need to capture the functional rôle model of each component — this relation seems less intuitive than a more domain specific behavioural model of the component and as the functional rôle is specific to the system, the resulting component model is less reusable than a behavioural model

might be. The analysis for FMEA is complex, as the effect of the introduced failure is traced all the way through the system's component models, but it does support tracing the effects of failures through mixed domains.

Another approach with some similarities to (Hawkins & Woollons, 1998) is that of (Chittaro *et al.*, 1994). In common with most other work on functional modelling, function is associated with components. Functional knowledge is represented by three models, a functional rôle model that interprets the component's behaviour, which participates in a process model, and the processes, in turn, participate in the model of the phenomenon. The teleology of the system is defined as its goals, which are assumed to be achieved by the phenomenon model. This means several models are needed for components, as each has a structural, behavioural and functional model in Chittaro's "multimodeling" approach. There is more on this in section 4.2. A theoretical basis is claimed for this method.

The Multi Flow Modelling (MFM) approach of (Lind, 1994; Larsson, 1996) also has the idea of functions contributing to achieving goals. In this approach, which is intended for monitoring and diagnosis of industrial plant, components realise functions and functions achieve goals. These relations are many to many as a component can contribute to several functions. Each component function is taken from a set of function definitions intended to specify that component's effect of a flow of energy, material or information. This relates the means — end relations that exist between functions and goals and the whole — part relations involved in modelling complex plant. The MFM model of a plant is a static graph representing these relations. This has the advantage that the approach is scalable, the complexity of the graph increases linearly with the size of the model (plant). As this graph is the only model of the plant, the correctness of the results depend on the correctness of the model. The approach is also limited to its intended field.

These approaches to and uses of function contrast with its use in (Price, 1998). Here, the functional modelling is done at system level, by attaching system functions ("functional labels") to the output of significant components. For example, in a car lighting system the function "headlamps on full beam" is linked to the output of the two headlamp bulbs. Function is therefore defined in terms of purpose of the system. The functional label is easily added to a component centred representation of the system. Not every component has a functional label, typically they need only be attached to those components that generate the system's output. Three advantages are claimed:

- Simplicity, functional labels are easily added to structural representations especially as they relate so closely to the system's purpose.

- Re-usability, the functional label is not tied to how a function is achieved, so can be applied to some other design of system with the same purpose.
- Capability, The approach is capable of recognising the unexpected achievement of a function.

Linking function to the system avoids the problem of an associative representation of function relying on a one to one mapping between function and component. There is no difficulty in combining outputs of several components, if necessary, as in the headlamp case. The paper does not discuss the use for mixed domain systems, but as the function is linked to system output, the domains in the system are scarcely significant. This leaves a rôle for behavioural modelling as the basis for reasoning about the system, as described in Section 4.1.2. The relations between models in this approach and elsewhere is discussed in section 4.2. The usefulness of this approach has been demonstrated by its use in *AutoSteve*, the commercial tool resulting from the work. One difference between this representation of function and many others is that there is no explicit representation of the input of the function. This is because the input associated with a function can be derived from the simulation of the correctly working system for failure analysis. For design verification (such as Sneak Circuit Analysis), the input does need to be specified. The lack of representation of the input for a function is problematic when describing cases where a system has functions (such as telltale or back up functions) that depend on the achievement or otherwise of some other function. These cases are discussed in Chapter 10.

This approach merely uses the functional labels to interpret the results of a system simulation derived from knowledge of behaviour and structure, as discussed in Section 4.1.2. This contrasts with approach taken by the functional reasoning community, where functional knowledge is used as the basis of the reasoning process. Indeed, as the work at Aberystwyth progressed the functional models became simpler, as discussed in (Pugh *et al.*, 1995) and (Price & Pugh, 1996). This simplification of functional models started when it was found that the functional reasoning approach duplicated the results obtained by the domain simulator, so building the required functional models was found to be redundant.

An extension to that approach to functional modelling has been suggested by (Snooke & Price, 1998). In this variation, each significant component has a functional label attached and the system's functional model is built up from the component functional models. To reuse the headlamps example, instead of the function "full beam" being associated with the left and right hand lamps being active, each lamp has a functional label for full beam and the system function "full beam"

is achieved if both functions “left full beam” and “right full beam” are achieved. This has the advantage of adding detail to the explanation, along the lines of “function full beam not achieved because left full beam not achieved”. Against this advantage is the need for some additional work in adding the more complex functional labels. This extension seems to provide a possible starting point for the idea of associating functions with subsystems for modelling mixed domain systems.

A distinguishing feature of both (Price, 1998) and (Milde *et al.*, 1999), is how much less use they make of functional representation, compared to the functional reasoning community. They both use behavioural models in a component centred representation, combined with domain rules, as the basis for reasoning. One possible explanation is that both systems are concerned with modelling faulty behaviour, which will not readily be captured using component level functional modelling, although Hawkins and Woollons use functional modelling for a similar task. Another arguable point is that behavioural models more readily capture the available knowledge, so are easier to use. For example, the user (an engineer) can readily use an existing behavioural model of some component, say an electric motor, and this can be used in simulation without asking the engineer to provide a specific description of the motor’s function in the context of that system. Both these systems are intended for use by engineers, and maybe making fuller use of functional modelling, at least in this context, is merely unnecessary complication. Despite this, it is tempting to suggest that functional modelling is more applicable for other tasks, such as recognition, that is deriving function from structure, using functional models of known components. The corresponding drawback is the limitation on the range of systems that can be analysed because of the reliance on domain rules. The avoidance of domain rules is one of the claimed advantages of a functional approach.

The functional labelling approach resembles the association of function and system behaviour in (Sembugamoorthy & Chandrasekaran, 1986). There function is defined as the what a device’s response is to an external stimulus and behaviour explains how this response is achieved. The approach links a functional label to an external stimulus (such as pressing a button) and a set of descriptions for the system behaviour. The behavioural descriptions explain how the stimulus achieves the function for a correctly working system. A drawback is the difficulty of building the behavioural descriptions for large systems.

There is an alternative approach to associating function with output in (Chandrasekaran & Josephson, 1996). Here, the function of a device is simply defined as its effect on its environment. It is expressed in terms of relationships in the

environment, so a pump, for example, will have its function defined in terms of volumes of water at different locations in the environment. This is intended as a general definition of device function, capturing various intuitions. An important part of this definition is the idea of a "mode of deployment", which defines a causal interaction between the device and the environment. They suggest a primitive representation of an "object in an environment, viewed from a perspective". The view supports abstraction of composed objects. The paper imagines a design activity that proceeds from a library of stored designs, specialising the designs and composing items from the library. This relates closely to the purposive definition of function in (Chittaro & Kumar, 1998). The paper fails to demonstrate their notion of explaining a device level function in terms of component functions, which is a feature of the hierarchical functional reasoning proposed by (Snooke & Price, 1998).

Work has also been done on using functional reasoning to support the design process, notably by (Iwasaki *et al.*, 1993). Here function is defined to mean the intended behaviour of the system (or device) and as such includes notions of purpose. The model of the design process follows the idea in (Gero, 1990) that the physical structure of a design is derived from a causal mechanism and has the functional model of the system being refined and so informing the refinement of the physical design. The physical specification must include all components specified by the functional specification and the requisite connections. It is not made clear how the physical specification is represented, however, and nor is it made clear how the system verifies the design — is the system behaviour derived from its physical structure or from the functional model? There seem to be problems with the simple model of the design process as in many cases, at least, some of the physical design might be derived from knowledge of other designs (case knowledge) and functional refinement will not be necessary. These points are discussed further later.

There is a slightly different definition of function in (Iwasaki *et al.*, 1995), where function is defined with reference to a goal, rather than to intended behaviour. The definition of function is given as

> An object O has a function F if there is an agent who can use O under some circumstances in some specific manner to achieve a goal.

The paper also defines a "device" as a physical object that has a function. There seems to be no reference to the idea of intent in this definition. For example a handyman could use a chisel to lever open a paint tin and this appears to suggest that, according to the definition, a chisel has that rôle as a function. However,

intuitively this is not the case, as this rôle amounts to abuse of the chisel, and will ruin the edge, rendering it less fit for its intended function. There does seem to be a need for some more explicit notion of intent in this definition, though the lack is perhaps understandable in the context of supporting the design process, as in this process, unanticipated uses of a device are more or less by definition excluded.

An approach to supporting the functional design of a device was introduced in (Umeda & Tomiyama, 1993). That paper divided the design process into three stages; functional design, basic design and detailed design. In this case, functional design is concerned with what a device must be able to do, so is related to the needs it must fulfil. They define function in terms of human abstraction of behaviour and treat it as subjective, as opposed to the objective nature of behaviour, which is determined by the underlying physics. They use a model of function, behaviour and state, in which function is related to behaviour and behaviour is defined in terms of changes of state over time. It is therefore not clear how static (as opposed to dynamic) functions are modelled, though they could perhaps be modelled in terms of the absence of undesirable changes of state. This approach has been demonstrated for designing increased reliability into a system by introducing "functional redundancy" (Umeda & Tomiyama, 1993) and for designing products so as to allow easy upgrading of the products in future (Umeda *et al.*, 2005).

There is a discussion of these definitions and uses of knowledge of function, with a proposed definition to be used for the remainder of this thesis in Section 4.1.5.

4.1.4 Teleological knowledge

In the context of design analysis, teleological knowledge is generally taken to be knowledge of the purpose of the system or component. The idea of purpose can introduce difficulties in modelling certain natural systems. While there seems to be little difficulty in stating the purpose of the heart as driving the circulation of blood, there seem to be philosophical difficulties in using the idea of purpose in an ecological system. For example, the behaviour of a predator will certainly include the killing of its prey, and that will have the effect of controlling the prey's population but unless one accepts the theological doctrine of teleology (that there is evidence of design in the natural world), then considering the control of population as the purpose of the predator seems problematic. It is not proposed to explore this area of discussion any further here, except to suggest that this is one reason why these four classes of knowledge are more appropriate for modelling of engineered systems, the subject of this thesis. (Chittaro & Kumar, 1998) suggests

that some researchers feel the need to differentiate between the idea of function and the idea of purpose. They differentiate between the definitions by associating purpose with human notions of utility in a sociocultural context. In other words, teleological knowledge is concerned with the need the system is to fulfil, such as the need to light the road ahead to allow a car to be driven after dark.

If teleological knowledge is concerned with the purpose of a system (or component) this does seem to leave an overlap with the purposive idea of function. It can be argued that the question given in (Chittaro & Kumar, 1998) as answered by functional knowledge ("What does a component do?") and that answered by teleological knowledge ("Why is a component in the system?") differ as much in their respective emphases on the component and system than in their actual content. For example the question "what does a safety valve do?" could have the answer that it prevents a boiler explosion by relieving the pressure" while the teleological question "why is a safety valve in the system?" might have the answer that it is to prevent a boiler explosion, which it does by relieving pressure. This is not to suggest that there is no room for both classes of knowledge, rather to attempt to illustrate how similar they can be considered to be. The requirements of a system might be considered either in terms of purpose or function. The functional decomposition approach of (Iwasaki *et al.*, 1993), applied to the model of the design process in (Gero, 1990) could start out with a stated requirement that is a description of purpose, rather than of function, which is then itself decomposed into more specific, closely defined and formally represented functions. This is illustrated in Section 4.2. For example, one might begin the functional refinement of the design for a washing machine with the simple (not to say obvious) statement that its purpose is to clean clothes and break this down into more refined functional specifications, such as the need to wash with soapy water, then rinse then remove at least some of the water. This refinement will continue in the manner proposed Iwasaki's paper. This is discussed further in Section 4.2. It might be argued that this is quite a useful model of the process of requirements capture for a new system, the aim being to reify the original (typically informal) idea of the purpose of a new system to generate a sufficiently well specified set of requirements that the success or failure of a system in meeting the requirements can be assessed.

(Franke, 1993) differentiates between purpose and function by defining a teleological description as concerned with "the elements of the structural ontology considered in the context of requirements or specifications of the mechanism or system" as opposed to function being considered in the context of the (component's) surrounding system. This seems to suggest that a functional description is

applied to an element in a larger system (a component or subsystem) while a teleological description is applied to the system itself. As the paper suggests that the teleological specifications are expressed in terms of structure and behaviour, this tends to reinforce the similarity between notions of function and purpose. This is taken a stage further in (Kaindl, 1993) where a functional description reveals the intended purpose of a component or connection, so the function of a steam-release valve (to use Kaindl's example) is to prevent a boiler explosion. This certainly seems to leave no room for a more abstract teleological description (that paper does not include purpose as a separate class of knowledge) while being consistent with the definition of function in (Franke, 1993) in so far as the valve's function is in the context of its surrounding system, the boiler.

A language for representing teleological descriptions was proposed in (Franke, 1991). This represents changes to a system design in terms of changes to the structure and changes to its behaviour. To use the example given in the paper, the addition of a pressure relieving valve to a boiler changes the behaviour of the boiler such that its internal pressure will not exceed some known value. It is tempting to argue that this is closer to Franke's later definition of a functional description, being, as it is, placed in the context of the system. It is also devoid of any explicit notion of the purpose of the component, in this case the prevention of boiler explosions. This seems to offer further evidence of how closely linked many researchers' notions of purpose and function appear to be. This language makes use of primitives to describe the change the new part of the design has its behaviour, which looks not dissimilar to the functional primitives in some functional description languages. However, the use of these languages does appear to differ, as the component based functional descriptions are intended to build a causal description of the system's behaviour. This does not appear to be the intention with this teleological description language.

While the foregoing appears to suggest that there is little room (or perhaps need) for teleological representation as well as functional, one area where a notion of purpose is of value is in the generation of design analysis reports, such as FMEA reports. This is simply because the consequences of a system failure appear in these reports and these will relate to the intended purpose of the system. Therefore some knowledge of the purpose of the system is required if these reports are to be generated automatically.

As the idea of function is associated with a system in the functional labelling approach of (Price, 1998) there might be thought to be little room for any more abstract notion of purpose, especially in the light of the proposed definitions in (Franke, 1993). Arguably a functional label has no formal notion of purpose, but

as a design analysis report generated automatically using functional labels will include consequences of failure of a system function, this at least will typically refer to an implicit notion of purpose. For example, a car headlamp system might have as a consequence of failure that the road ahead is not lit, implicitly referring to the idea that the purpose of the system is to light the road ahead. This area is expanded upon in Chapter 5 but it is worth pointing out here that there might be additional information included in a functional label, such as the consequence that it is illegal to drive the vehicle if the headlamps have failed. This points to the idea that functional and teleological knowledge, especially when used for interpretation of behavioural simulation of engineered systems provide a way of capturing knowledge that is external to the system and therefore beyond the range of the simulation engine itself. There is a discussion of the relationship between the classes of knowledge, the boundaries of the system under analysis and the boundaries of the simulated world in the following section.

As a final note on the rôle of teleology, it is worth pointing out that there has been work in qualitative reasoning whose aim is the recognition of the (previously unknown) purpose of some system or device. This was an aim of (de Kleer, 1984), in which teleological reasoning was employed to derive knowledge of some system's unknown purpose from knowledge of its behaviour. As the present work is concerned with design analysis this use of teleological reasoning is outside its scope, as it seems safe to suggest that any design process will be founded on knowledge of the purpose of the system being designed.

4.1.5 Using the four classes of knowledge

The earlier sections in this chapter have been largely concerned with previous work that has made use of some or all of these four classes of knowledge. It also attempted to identify common threads in these (frequently contrasting) approaches to functional modelling. This section will propose a set of definitions of these classes of knowledge that will support the intended use of functional knowledge to support the interpretation of simulations and allow the automatic generation of textual design analysis reports. How these new definitions might relate to other researchers' approaches will also be discussed. The focus is now, therefore, less on other researchers' work and rather lays foundations for the material in the succeeding chapters, describing the Functional Interpretation Language that is one of the main results of the present research. However, the present chapter will conclude with a discussion of the relationships between these classes of knowledge in other work, in the light of the proposed definitions proposed here, concentrating on the

uses of functional knowledge.

4.1.5.1 Proposed definitions of knowledge classes

This subsection will propose a set of definitions of the classes of knowledge and suggest a relatively well defined way in which they can be considered to relate to each other. The definitions and relations will be explained with reference to possible examples to attempt to clarify the differences between them, specifically between function and behavioural knowledge and between function and purpose.

Informally, the four classes of knowledge might be defined as follows:-

Structural knowledge is concerned with the elements that make up a system and the physical relationships (connections) between them. Typically, this might be concerned with physical components and connections but the elements might be processes. For example, a torch contains a battery connected to a switch connected to a lamp connected back to the battery.

Behavioural knowledge is concerned with what takes place within a device. This will typically be the possible states of a device and the possible transitions between them. The torch can be on (switch closed and lamp lit) or off (switch open and lamp not lit).

Functional knowledge is concerned with how a device fulfils its purpose. For example what system states can be associated with fulfilling some purpose. The torch being lit can be associated with the purpose of enabling a user to see in the dark.

Teleological knowledge is concerned with the requirements (or needs) a device is intended to fulfil. The torch is needed so that a user can use it to light their way in the dark.

The term device has been used to attempt to free these definitions from any specific notion of what level of system decomposition they are associated with. For example, if a system is analysed using a qualitative simulation, knowledge of the system's behaviour might be derived from knowledge of its structure and of the behaviours of its components. The same definition of behavioural knowledge should be applicable to both component and system behaviour.

One approach to clarifying the relationships between these definitions is with respect to the device itself. It is suggested that both structure and behaviour are internal to the device and purpose is external to it. Function, being related to

both behaviour and purpose is concerned with describing the interface between the device and its environment, in terms of purpose. It can be thought of as an external ("black box") view of the device and to be related to the device's environment (which might be some surrounding system or the environment in general). To illustrate, consider an electric torch. The purpose of the torch is to allow the user to see in the dark. This is external to the torch itself, notice that the need could be filled with a lantern or candle. The functional view of the torch reifies this need by describing how the torch can be used to fulfil the purpose, in this case by the switching on of the torch resulting in a beam of light. The behaviour of the torch describes how switching on completes a circuit linking battery and lamp so the lamp has current flowing through it and the structure describes the components and connections that make up the circuit. A not dissimilar distinction between behaviour, function and purpose is drawn in (Franke, 1993).

As the present work is concerned with interpreting the result of a (behavioural) simulation with respect to purpose, it seems worth proposing a more formal definition of function than that listed above. The following is suggested.

> Function: An object O has a function F if it achieves an intended goal by virtue of some external trigger T resulting in the achievement of an external effect E.

This is similar to the definition in (Iwasaki *et al.*, 1995) but specifies that the goal is intended (by the designer) and the adds the notions of trigger and effect, which are discussed in the following chapter. For example, a torch achieves the goal of lighting the dark when switching it on triggers the effect of the lamp shining. The nature of the trigger and the effect will depend on the nature of the device whose function is being described. This approach to function can also be used at the component level (assuming the use of a component centred ontology) so that the function of the motor in an electric fan is to generate a flow of air, triggered by current flowing through its windings leading to the effect of the shaft rotating the fan blades. This adds some notion of purpose (which is specific to the system that includes the motor) to the description of behaviour of the motor. The trigger and effect of the function are defined as being external so as to express the idea that the function of a system is concerned with its interaction with its environment. The environment might, of course, be some larger encompassing system that is being treated as beyond the scope of the present analysis.

This definition raises the point of whether a functional representation should incorporate a specific description of the goal (purpose). It is suggested that there

are advantages in the idea that it does not, but rather has a reference to a teleological description that is treated as a separate component of the model. In other words the model incorporates a teleological description *achieved by* a function, which describes how the purpose is achieved. This is discussed in more detail in Chapter 5 but the point is worth raising here as this style of representation has a bearing on the illustrations of uses of functional knowledge illustrated in the next section.

The idea that behaviour is local as against function relating to the device's environment raises some interesting points about what can be included in a device's behavioural model where a device is modelled as an atomic component with a behaviour rather than in terms of its structure and underlying domain based behavioural rules. Such a behavioural description should make no assumptions about the state of a device's environment. For example, rather then describing its structure and deriving its behaviour from domain rules, an electric motor's behaviour might be described informally as "if there is current flowing through the windings, then there is torque at the shaft". This contrasts with a motor's functional description which will (almost certainly) be concerned with moving something. This does, of course, make assumptions about the motor's environment, most obviously that its shaft is connected to the component it is to move and that the connection allows the required movement. So the motor of an electric fan, for example, might be behaving correctly in that there is current flowing through the windings and there is torque at the shaft, but it is failing to achieve its function (making the fan blades rotate) because one of the blades is fouling the casing, perhaps. This arguably raises something of a difficulty with the idea of using function in the simulation of a system's behaviour, as whether a component is actually fulfilling its function can only be established with reference to the behaviour (function?) of the rest of the system.

Arguably a better example of this difference is provided by a digital component, if only because there is no benefit in modelling such a component at a structural level, its behaviour is almost certainly best described in terms of the logic it implements, though this does assume that the logic is implemented correctly.

4.1.5.2 On the proposed definition of function

Having introduced a new definition of function, it seems worth including a discussion of how the definition relates to other researchers' definitions and uses of the idea of function.

It is, perhaps, arguable whether this is beneficial, but one feature of this definition

of function is that it unites the purposive and operational notions of function in (Chittaro & Kumar, 1998). On the one hand it relates behaviour to teleology by defining the external aspects of its behaviour (the trigger and effect) that fulfil the purpose and on the other hand it relates a trigger (which might be a representation of input) to the (expected) effect, which equally might be thought of as output, though it might, in many cases, be better considered to relate to a goal state.

This observation suggests that the proposed definition of function incorporates the elements of other definitions, so other definitions can be regarded as aspects (or components) of the proposed definition. Therefore the intended goal can be likened to the definition of function as purpose in (Kaindl, 1993) and the trigger and effect can be thought of in terms of input and output, consistently with (Navinchandra & Sycara, 1989). The relationship with the definitions of function and teleology in (Franke, 1993) raises some interesting questions. His definition of function as being in terms of "the behaviour of the larger, enclosing mechanism or system" could be seen to imply that function belongs to a component or subsystem, while the definition of purpose as being in terms of the requirements of the system places that, perhaps, a little more at the system level. There is arguably some common ground between Franke's definitions and the idea expressed above that behaviour is internal to the device, purpose external and function in between, at the boundary between the device and its environment. The suggested approach avoids the apparent implication that function depends on some larger mechanism, though this objection is weakened if one accepts the idea that a system's environment is treated by the reasoner identically to a larger enclosing system. This is (arguably) unlikely, simply because the reasoner will not contain sufficient knowledge to model the environment in general in the same way as the system under analysis. In the absence of a reasoner having a complete knowledge of physics (qualitative or otherwise), either the reasoning (simulation) will be domain based, restricting analysis of the environment, or if a functional reasoning approach is used, the network of functional relationships must be extended to include necessary aspects of the environment.

These relationships raise the question of the relationship between the idea of the system under analysis and the simulated world. One important rôle of a functional representation is, at least for the present work, to capture knowledge about the world around the system being analysed. The idea proposed above that purpose is external to the system implies this, assuming that the system being analysed corresponds to the simulated world. For example, we might state that the purpose of a car headlamp system is to light the road ahead, and refine this into a functional representation in which this purpose is achieved when the

headlamp switch is turned on and the headlamps light. This now allows the system's simulation to be interpreted in terms that relate to the world outside the system so the simulation engine itself needs no knowledge of them. All the simulation has to show is that the headlamps will indeed light up when the switch is turned on and the functional description will capture the (implicit) knowledge that this will light the road ahead, assuming that it would be dark otherwise. This is, it will be remembered, what the "functional labelling" approach of (Price, 1998) already does.

The present definition of function is distinct from notions both of behaviour and purpose. This is not always the case in definitions and uses of function in the model based reasoning and functional reasoning communities. For example in the way (Kaindl, 1993) has function synonymous with purpose or the way that the functional reasoning community use (component) function instead of (component) behaviour, as in (Hawkins & Woollons, 1998).

This proposed definition of function can be regarded as an attempt at a more closely defined idea of function than that used in (Price, 1998). A more detailed comparison is perhaps best left until later, but it is worth pointing out that in the original functional labelling approach (for FMEA), function was implicitly defined largely in terms of output, or effect. The definition proposed herein attempts to extend that by including express knowledge of input (or trigger) and purpose. How these fit into the representation of function itself will be discussed in Chapter 5.

It seems well worth considering how well the proposed definition of function fulfils the desiderata for a representation of function given in (Chandrasekaran & Josephson, 1996).

Arguably, the first desideratum, that a representation of function should "apply to intended functions of human designed devices and to functions or rôles in natural systems" causes difficulties for any purposive definition of function, including that proposed. These were introduced at the start of the discussion of teleological knowledge, Section 4.1.4. In other words, any notion of intent implies a notion of a creator of the system with intentions for it. While replacing the word "intended" in the proposed definition with "expected" might allow its use for some natural systems, such as in specifying the function of an organ, it is arguably inappropriate to use any idea of a goal in ecological systems. Reducing the population of prey is an effect of predation, but not its goal. Indeed it is disadvantageous to the population of predators. As the present work is concerned with design analysis of engineered systems, this objection to the proposed definition is not important to this research.

It might be felt that the proposed definition of function does not fit the second of the three desiderata, that a representation of function should "apply to functions of both static and dynamic objects (e.g. support)". However, this need not be the case, note that (Chandrasekaran & Josephson, 1996) already uses the idea of effect in defining function in a way that is argued to fulfil this desideratum. The notion of a trigger does, it is appreciated, carry connotations of movement or change but there seems no real difficulty in specifying the trigger as a static entity. To use the idea of an entity having a function of support, the trigger can be represented in terms of the maximum intended loading of the entity and the effect as the absence of movement in the structure. A plausible example of the representation of a static function might be that the goal of a girder is to ensure the strength of a bridge, which it achieves adequately if a loading of x tonnes triggers a deflection of no more than y millimeters. The effect is the lack of excessive deflection under the designed load.

The final desideratum, that a functional representation should "apply to functions of both abstract and physical objects (e.g. software modules or steps in plans)" seems to present no difficulty. The goal of the entity can be represented in such a case, the trigger is readily defined and likewise the effect. For example, the goal of carrying out an FMEA early in the design process is to find how safe a system is when operating under component failures so as to establish whether changes should be made to the design. The trigger is completion of a candidate design for the system and the effect is the acquisition of knowledge to be used to inform the decision on changing the design, achieved by production of a report listing the effects of component failures. Whether such a full representation of function is useful in this case is, perhaps, open to question and a definition of purpose (that is, the goal) might be adequate.

One possible difficulty with this approach to function, at least in some areas, is that it is perhaps unnecessarily detailed. It is clearly more elaborate than (Kaindl, 1993) which defines function purely in terms of purpose.

4.2 Relationships between the classes of knowledge

In this section, the definitions and suggested contexts of the four classes of knowledge discussed above will be used to compare and contrast the uses of different knowledge by different researchers in the field. In view of the present work's emphasis on functional knowledge, the focus in this review will be on uses of that

class of knowledge. In view of the fact that other representations of function do not include all the elements expected by the more formal definition of function given above, it is perhaps more useful to use the informal definition, so in this review functional knowledge is classified as requiring some notion of purpose and also some representation of how this purpose is achieved, but that representation need not be in terms of triggers and effects.

Earlier in the chapter the idea that the different classes of knowledge can be placed in a hierarchy, with each class being an abstraction of the next class below it in the hierarchy, was introduced. For example, a component behavioural model can be thought of as abstracting the component's structure. Another hierarchy we can consider is that of system elements. This can be thought of as having four layers.

Product The top layer in the hierarchy, having several distinct (and largely independent) functions. For example, a motor car combines several quite distinct systems that either contribute to its core functionality (such as the transmission) or are necessary adjuncts, such as lighting.

System A self contained set of related components (or subsystems) whose role is to fulfil one aspect (or several closely related aspects) of the product's functionality. The lighting system of a car would be a good example.

Subsystem or module An identifiable part of a system, identifiable either in terms of physical separation (the front lamps of a car lighting system) or in terms of function (such as the direction indicators in the car's lighting system). While the term subsystem is frequently used to refer to a physical subset of the system and module to refer to a functional subset, these terms are treated as interchangeable.

Component An individual item. For the purpose of modelling, we can regard a component as being something to be modelled as a single atomic unit, the bottom level of system decomposition.

It is not impossible that there are several intermediate 'subsystem' layers between system and component, depending on the complexity of the system and how this system is to subdivided. Another possible decomposition, typically at system or subsystem level, might be in terms of domain. For example, a domestic washing machine might have its structure decomposed into water and electrical systems. It will be appreciated that in simple cases, the product layer will be unnecessary as a simple product can be regarded as embodying one system, which might be decomposed into subsystems. The washing machine is a case in point.

This decomposition of a system can be regarded as orthogonal to the idea that each class of knowledge in the hierarchy introduced in Chapter 4 abstracts the previous class of knowledge. It should be noted that there are problems with the treatment off these hierarchies as being orthogonal, to be discussed later. This introduces the idea that we can use a simple two dimensional grid to represent the space in which the necessary models are placed. This grid can have the system hierarchy as the vertical axis and the four classes of knowledge as the horizontal axis, as illustrated in figure 4.1. The grid might be used to show the models a user

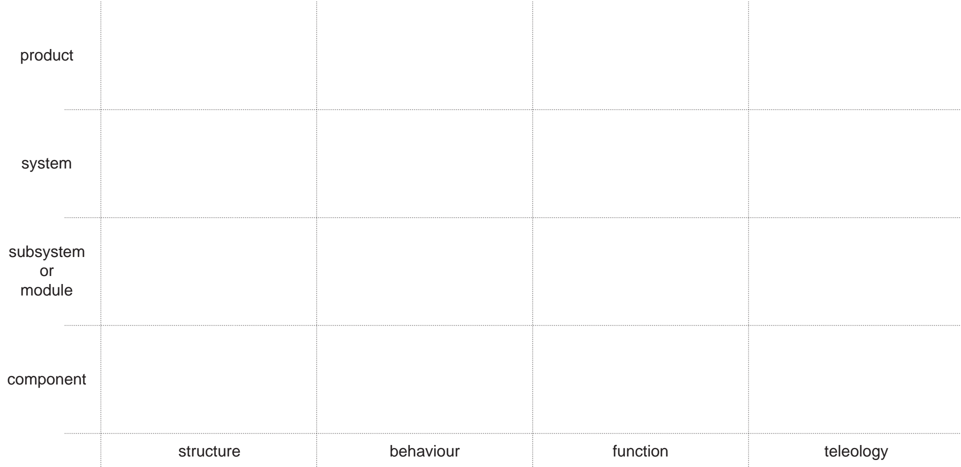


Figure 4.1: Grid to show possible model relationships

is expected to provide and also the models the system itself will derive, and can also show relations between models, either expressed by the user, or implicit in the system. In these diagrams, the models the user will have to provide are shown as grey boxes and the models either included in the system (such as domain models) or derived by the system's reasoning as white ones. The relations the user should explicitly express are shown as solid arrows and those native to the reasoner are dashed arrows. The grid in use is illustrated in Figure 4.2 on page 79 and there is a key to the symbols used in Appendix C on page 285.

In any given case, it is unlikely that all the layers in the system hierarchy will be used. Indeed the typical, simple, case is that a system will be modelled as being assembled from components. It does seem possible, however, that during the design process, small systems will later be combined as subsystems of a larger system and so on up to the product level. It seems likely that in most cases the aim of the reasoning (certainly the simulation) will be to establish the behaviour of the entity that is currently the highest in the hierarchy, possibly using the results of earlier simulations of smaller systems to allow subsystems to be treated as atomic components whose behavioural models are used in simulation of the newly

integrated larger system (or product). Work has been done on this approach, see (Snooke & Bell, 2002), with the aim of simplifying simulations of systems whose size might otherwise render them intractable. The only layer, therefore that has a fixed nature throughout the process is the component layer, as the diagram is based on the idea that a component is treated as being atomic and is modelled primarily in terms of its behaviour (or function). As has been observed, some representation of component structure might be needed to complete the structural description of a system for a domain based simulation engine, such as CIRQ.

It is also frequently the case that a specific approach will not use all the classes of knowledge either. Indeed, one of the aims of the functional reasoning community is to reason with a minimal set of knowledge. It is therefore the case that not all the spaces in the grid will be filled. How different researchers' model sets fit in with the grid will be described later in the section.

Implicit in this hierarchy, from the point of view of modelling, is the idea that any layer except the lowest (component) will need a representation of structure, so that that element's behaviour can be determined from the behavioural models of the next layer down, and its structure. There remains a decision to be made as to what level of complexity we wish to place at the bottom of our hierarchy, represented in terms of behaviour (or function) rather than structure. For example, do we treat an electric motor as a component, simply modelling it with some sort of behavioural model (such as a state chart) or is its internal structure to be modelled, allowing its behaviour to be derived from its structure and the necessary domain laws from a suitable qualitative physics? The answer will depend on the nature of the task to be undertaken, of course. An engineer designing an electrical system is likely to be happy with a simple behavioural model of a motor, treating it as a component of his design.

This leads to a characteristic of what is meant by a component in this context being that it is modelled in terms of behaviour rather than structure — so is at the bottom of the decomposition. A similar characteristic of the system level on the hierarchy is also useful, as it is the top level of our decomposition, if it is to be modelled usefully, it should be the case that the closed world assumption is applicable so we can ignore influences on its behaviour from anything outside the system. Any model reasoner will therefore use models at each of these layers but the other two layers need not be used in a given case. The subsystem layer might have models included to simplify management of the information (for example, having a structural diagram split into separate subassemblies) or for functional decomposition in that column of the grid. These two uses of the subsystem layer mean that there will not necessarily be a mapping between subsystems in different

columns. An example is the way that the transmitting and receiving functions of a telephone both use components in the chassis and handset. The product layer has been included largely for the sake of completeness. The main reason for it might well be to allow several systems to be combined to analyse their interactions, especially as this is a common cause of sneak circuits. It could, however, be argued that once this is done, then the product becomes the system under analysis, so this layer shifts. As noted above, (Snooke & Bell, 2002) describes an approach to deriving a behavioural model of a system or subsystem and substituting this for the structural model. This is intended to avoid the size of the system becoming intractable for running design analysis simulations on a whole product (such as a car) and this could be seen as a use of the product layer in the grid.

4.2.1 Relations between models within the grid

In the grid, there is a column for each class of knowledge, but it will be appreciated that not all columns need be used, depending on the approach taken. Indeed, one problem with this attempt to define the relationships between models is that the distinctions are by no means clearly defined, nor are they consistent between researchers. These relations will be discussed in general terms here and illustrated by considering the model relationships used by different researchers later in this section.

The aim of most of the approaches discussed in Chapters 3 and 4 is to generate a behavioural model of the specific system under consideration, to occupy the space at the top of the second column. How they differ is in what knowledge is used to derive this. Individual cases will be discussed later. It is true that all approaches have some knowledge of structure and the distinction between structure and behaviour is arguably the most clearly defined of those between adjacent columns in the representation. This is because structure is essentially static while all the other classes of knowledge are concerned with what things do and why, so are concerned with changes to the system's states and effects.

This structural model might be represented as a view of the system as a whole or by each model including knowledge of its neighbours. Which is chosen might depend both on how the reasoning is to be done, particularly whether a domain rule based reasoning engine is used and how the knowledge itself is to be captured, for example whether a schematic is used or whether individual components are arrived at by a process of functional decomposition. In the case of functional decomposition, the structure might be treated as implicit in knowledge of which functions contribute to which of the functions that are higher up the functional

hierarchy so there is no explicit representation of the system's physical structure, this being derived from the functional hierarchy.

As has already been observed, the view the user has of the structure need not be the same as that used by the reasoning engine. If, say, the system structure was derived from a schematic (so a global view) it would be perfectly feasible to use that information merely to instantiate each component model with knowledge of its neighbours. This might be useful in cases where a schematic covers several domains, not all of which have a suitable domain level analyser. Equally, it would be simple to use each component's knowledge of its neighbours to construct a global view of the system. If an electrical system had been specified using a functional decomposition, this would support the possibility of introducing a circuit analysis tool to analyse the behaviour of the system. This could be valuable if the circuit topology is complex, so that the difficulties in finding current through bridges that are associated with purely local structural knowledge apply in this case. One problem with this is that a global system structural model will need the internal structures of the components and this might not be available.

The relations between the other classes of knowledge are less clear cut. Indeed the functional reasoning approach might well omit the behavioural models and express components' behaviour purely in terms of its function. The operational definition of function as illustrated by (Hawkins & Woollons, 1998) can be regarded as mapping more closely to behaviour while the purposive definition provides a more explicit mapping between behaviour and purpose, especially when function is defined as intended behaviour, as by (Iwasaki *et al.*, 1993). This use of function could be argued to include notions of teleology. This will be discussed further in describing these sets of models.

It is tempting to regard the three right most columns in the grid as fairly arbitrary divisions of what is almost a continual gradation between representations of behaviour at one end and purpose at the other. However, this observation must be tempered by noting that as one of the most important tasks of design analysis is to establish how well the system's behaviour maps to its purpose, then in any given reasoning system there is a need for a clear cut distinction between the capturing of the knowledge of the intended behaviour and the knowledge that leads to the derivation of the actual (simulated) behaviour. As far as possible, models used in each illustration in this chapter have been placed consistently with the definitions of the four classes of knowledge proposed earlier. That set of definitions does provide a way of avoiding the danger of the divisions between the columns being too arbitrary.

There are other shortcomings of the simple illustration of model relationships,

which should be made plain, although it is felt that they do not prevent this matrix illustration being useful. The first of these is that there is nowhere in the grid to represent domain knowledge. There are arguments for placing it at the bottom, especially if it is used to underpin (and so explain) component level behaviour in the way "knowledge" is referred to at the bottom of the functional decomposition in (Sticklen *et al.*, 1989). However in the case of the domain level rules in (Price *et al.*, 1997) and the circuit reduction rules used by (Mauss & Neumann, 1996; Milde *et al.*, 1999) that are applied to the system, and as any such domain rule is applicable to other systems within the domain, it can be regarded as global in nature, so should arguably be added as a top layer. In both cases the domain model cannot really be considered a part of the system in the way component or subsystem models can, so it was felt reasonable not to allow space for domain models and add them at the top or bottom of the grid as seemed appropriate.

The other point to be made is that it could be argued that the axes of the grid are not truly orthogonal in that if a component's function is expressed as its effect on the system, then it could perhaps be argued that the relation between functional and behavioural model must imply moving both across the grid, from the behaviour to the function column, and up it, from component to subsystem (or system if there is no subsystem layer in the decomposition). It is considered that despite this, the grid remains useful. After all, a full functional decomposition will end at a layer where the behaviour is known and the relationship implicit in the representation of function as effect can be expressed as a link between the component and (sub)system level functional model.

Despite these points, the simple model of model relationships appears useful both in illustrating potential weaknesses in a specific model set and in showing how different model sets differ.

The commercial design analysis tool developed from earlier work in Aberystwyth is a useful starting point for comparison of different model sets as it happens to use a quite a large set of models. It should be observed that individually, these models are relatively simple and reusable.

The minimum set of models needed for FMEA by the design analysis tool developed at Aberystwyth is as shown in Figure 4.2. It will be seen that the user is expected to provide a representation for each column in the grid, except teleology. This might seem to ask the user for a good deal of work in input, but the component behavioural models are reusable, being independent of the system. The functional model is also reusable, being simply a list of functions the system is expected to achieve, these functions being related by boolean operators as described in (Snooke & Price, 1998). The system structural model is the schematic the en-

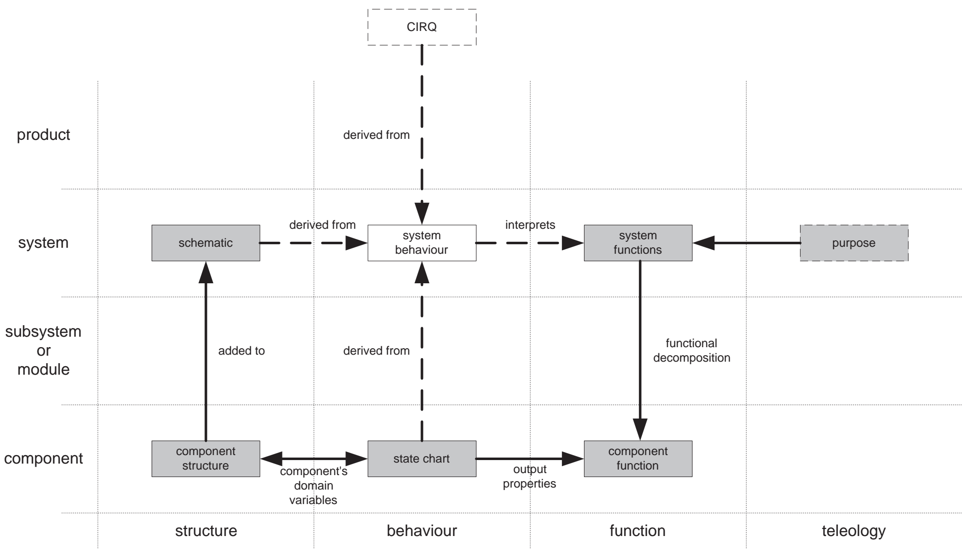


Figure 4.2: Minimum set of AutoSteve models for FMEA

gineer is working with, so that also comes with no additional effort. As a domain level simulator is used to define the relationships between the components, the components need internal structural models (shown at bottom left in the figure) to be combined with the schematic to create the network of resistances for CIRQ. These component structural models are also reusable.

The relationships in figure 4.2 are actually specific to FMEA. For sneak circuit analysis the sets of models required are identical (though the models themselves do differ slightly) but the relationship between the system behavioural model derived by the reasoner and the system functional model is different. While in FMEA, the functional model is simply used to interpret the results of a comparison between the system behavioural models of the correctly working system and the system affected by a failure, in SCA (and in design verification in general) the actual comparison is between the system behavioural models and the required behaviour represented by the functional model. This leads to the necessity of including the required inputs to the system (that is, the switch positions) in the functional model for SCA. This has been dispensed with for FMEA as the input states of the two behavioural models can be compared.

The models shown in Figure 4.2 are the minimal set that might be required. Typically, there will be functional models at the intermediate, subsystem, level, as the system functions will be decomposed into the necessary subsidiary functions, as described in (Snooke & Price, 1998). An example might be that a “headlamps” top level function might include subsidiary functions for side lights and tail lamps. These functions themselves can be decomposed into component level func-

tions leading to a one to one mapping between the bottom level of the functional decomposition and component behaviour. It could be argued that this mapping should be a requirement. This arises from the fact that it is possible to build an incomplete functional model and use that. This leads to misleading results in the FMEA report. For example it is quite possible to carelessly specify a headlamp circuit functional model in such a way that all functions are recorded as being achieved as expected, despite one of the headlamps staying on full beam the whole time. This leads to questions as to whether a component level mapping between behaviour and function will prevent this, and also as to whether a full functional model should specify what should not be happening (undesirable outputs to be avoided) as well as what should. One of the benefits of the Functional Interpretation Language described in this thesis is that it avoids this pitfall. This is discussed later in the thesis.

The "output property" [1] that links behaviour to function can be made to cross both between rows and columns in the diagram. This is done by using a Boolean expression to combine more than one output upon which a function depends as part of the output property rather than completing the functional decomposition. This is what leads to the possibility of the incomplete mapping between behaviour and function outlined above and leads to the suggestion that such "diagonal" relations between models in the grid should be avoided.

This set of models has been discussed at some length because of its importance as a starting point for the present work. The grid can also be used to illustrate the model relationships used by other researchers. This will now be done and both the models and their relationships compared and contrasted. The interpretation of the models used has, of course, been derived from the literature, as reviewed in the preceding chapter and earlier in this chapter. It was felt better to separate this material, partly as it was decided that introduction of different approaches was better separated from this more interpretive discussion, as interweaving the two might be more confusing for the reader and partly simply to emphasise that this material is interpretive, and the interpretation is my own, rather than the original authors', and others may prefer different interpretations.

Attention has already been drawn to the idea that there are two contrasting approaches to model based reasoning. These are a "bottom up" behaviour based approach (deriving behaviour from structure and component or domain behaviour)

[1] In *AutoSteve*, those components that are interfaces to the system have "interface properties" attached to them. These might be input properties, which allow the user to set input states of the system using them, so will be attached to switches, or output properties, which associate a component with a system output. For example, a lamp will have the output property "lit" associated with the state of having current flowing through the filament.

and the top down functional reasoning approach in which structure is derived from function. These contrasting approaches are illustrated by the different model sets used.

The model set described above, see figure 4.2, is consistent with a behaviour based approach, so includes structural and behavioural models as the basis for simulation and adds simple functional models for interpretation. A similar model set is used by (Milde *et al.*, 1999), as shown in figure 4.3. It will be seen that

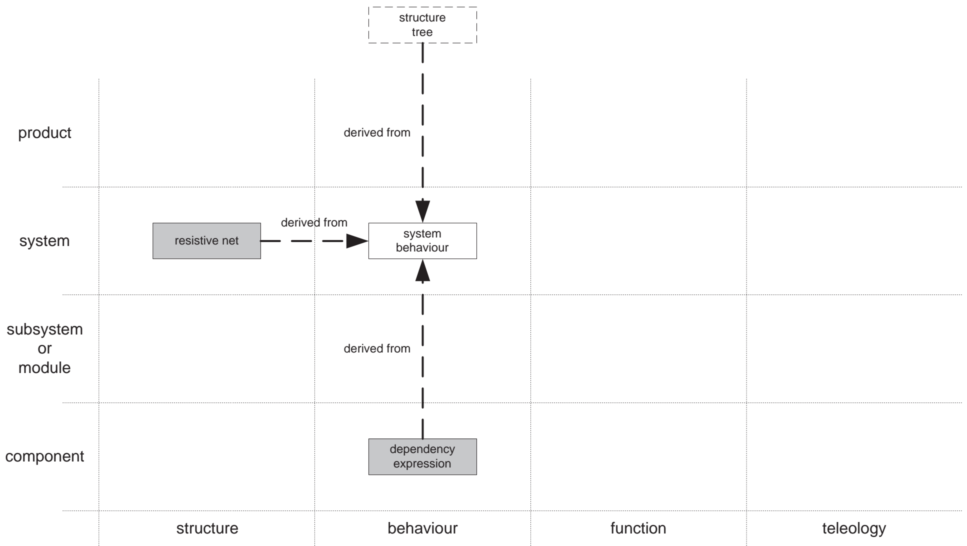


Figure 4.3: Models used in the behavioural approach

the set of models used for simulation is similar to that in figure 4.2, though the models themselves differ. The main difference is the replacement of CIRQ as the domain level simulator by their tree decomposition of the circuit. This does suggest that it would be possible to swap between these two alternatives relatively easily. How easily depends also on the models themselves being similar. The component models are similar, either can use dependency expressions (expressed as if...then rules) as are the structural models, as both represent the circuit as a network of resistances.

The most important difference is the addition of the interpretive functional models in figure 4.2, but the similarity of the models used for simulation does suggest that if it was desired, a similar set of interpretive models could be added.

It will be appreciated that the model set used by (Mauss & Neumann, 1996) is also similar, but as that paper does not explain how components with complex internal behaviour (such as relays) are modelled, that model will be missing. This would tend to militate against easy addition of functional models for interpretation of behaviour.

These model sets contrast quite strongly with the model sets used by the functional reasoners.

The functional reasoning approach has components' rôles defined in terms of their function in the system, so there is a functional decomposition down to the component level, as shown in figure 4.4. Here, the physical structure of the system

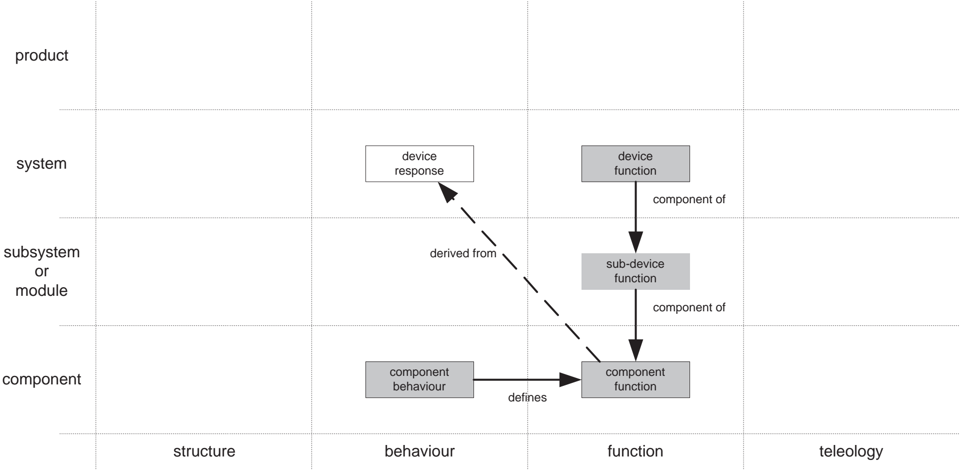


Figure 4.4: Models used in Functional Representation

is (implicitly) derived from the functional decomposition, which is underpinned by knowledge of components' behaviour. The functional models in the Functional Modeling approach (Sticklen *et al.*, 1991) actually incorporate knowledge of the structure (in that each component functional model knows which are its neighbours) and behaviour. This does perhaps raise the question of how appropriate this approach is for design verification, the aim of which is to compare what the system actually does with what it is intended to do. There might be thought to be a danger that if the composition of a system is defined in terms of function, the actual behaviour simulation will miss any possible errors in the physical specification. There is also the need to define the functional model for a component. If this is specific to the system, then such models are not reusable, and it might be argued that they are less intuitive for the user to build than the kind of context free behavioural models used by (Snooke, 1999) and (Milde *et al.*, 1999). However, this relationship between functional and physical structure does introduce the possibility of using knowledge of the functional structure of a proposed system to guide the design of its physical structure. This is consistent with the model of the design process used in (Iwasaki *et al.*, 1993).

The use of a similar functional decomposition is common to other approaches to functional reasoning. In (Chandrasekaran & Josephson, 2000), the top level function in a decomposition is described in terms of the device's effect on its en-

vironment. This is contrasted with the "device centred" functions lower down the decomposition, which are closer to abstract behavioural models, described in terms of variables that are internal to the system. For example, an electrical switch is defined in terms of the voltage across its terminals (if closed) and the absence of current through the switch if it is open. The examples quoted in (Chandrasekaran, 2005) are dependency expressions not dissimilar to the component behaviour models used in (Milde *et al.*, 1999). One slight difference is that a behavioural model of a switch might be expressed in terms of its electrical resistance as the behavioural simulator can use domain knowledge to find the switch's effect on the voltages and current in the circuit.

The models used by (Iwasaki *et al.*, 1993) are similar, but as their representation of function includes explicit notions of purpose and as they seek to model the design process, then there is a fuller set of models in both the functional and especially the teleological columns, as in figure 4.5. As their representation of function

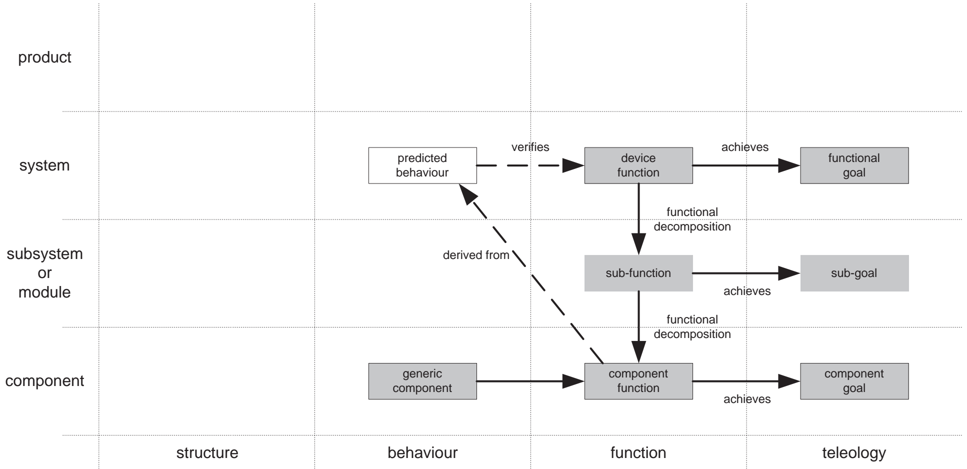


Figure 4.5: Models in CFRL

includes the function's goal, that suggests that part of the functional representation belongs in the teleology column of the grid. The additional (optional) models of subsidiary goals and functions are intended to represent their model of functional refinement as part of the design process. However, it is arguable that this model is an over simplification of the design process. A design support tool should, though, be able to offer support to all stages of the process, and so arguably needs to support the idea of functional refinement. The more complete functional model that should result would address the problem identified above with the functional labelling in figure 4.2, but if the tool is only to be used for behaviour based analysis of a system whose physical design is done, such a model might be unduly elaborate, and capturing the necessary functional knowledge might be excessively

time consuming for the user.

The multimodeling approach of (Chittaro *et al.*, 1993), figure 4.6, does include explicit representation of the connections between the components. The verification model of system behaviour is derived from knowledge of the components' functional rôles which are, in turn, derived from knowledge of the components' behaviour.

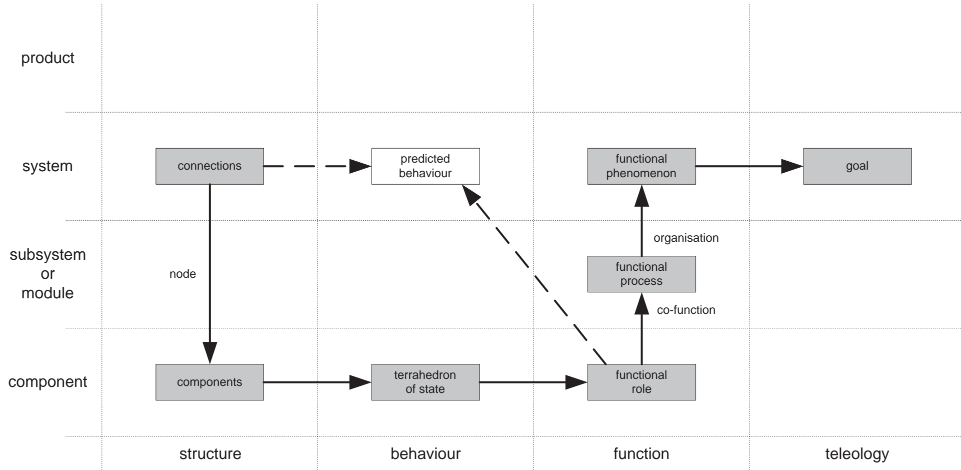


Figure 4.6: Models used in Multimodeling

It could be argued that the express representation of a component's functional rôle is unnecessary, as in many cases it can be implicitly derived from knowledge of its behaviour and the system structure. For example, the functional rôle of the wire connecting the left headlamp main beam pin can readily be identified as part of the path of current from battery to lamp, so its rôle is to transmit any flow. There is arguably no need for such elaborate functional modelling unless it is of help in refining the design, as in the model of the design process suggested in (Iwasaki *et al.*, 1993).

It will be seen that in all the cases illustrated above, the job of the reasoner is to generate a model of the system's behaviour. This contrasts with the aim of (Falkenhainer & Forbus, 1991) which is to generate a sufficient model to allow a specific query to be answered. It might be suggested that the aim of the reasoner in their work is to generate a behavioural model of a sufficient subset of the system to allow the query to be answered. In figure 4.7, this different aim has been illustrated by placing the reasoner generated model in the subsystem layer of the grid. This implies an interpretation of the aim of compositional modelling in terms of its identifying a subsystem whose behavioural model is sufficient for answering the query concerned.

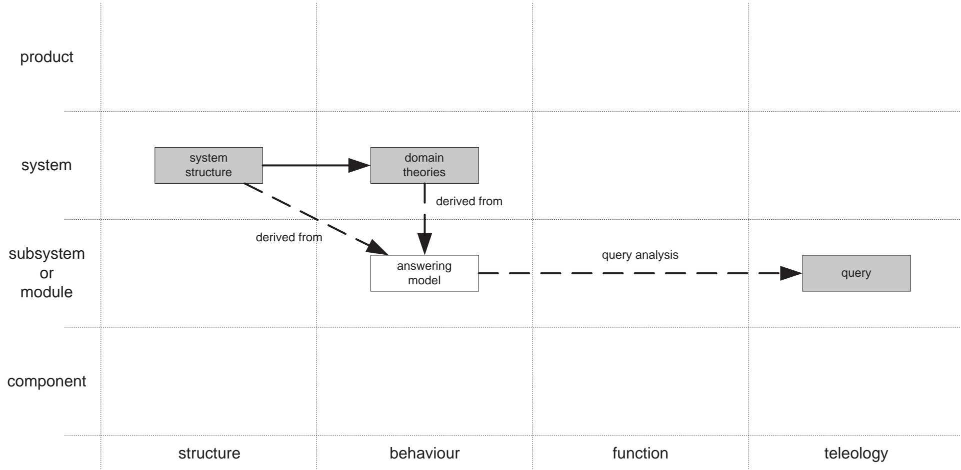


Figure 4.7: Models for Compositional Modeling

It might be suggested that the aim of the compositional modelling approach is better suited to other tasks than design analysis. The example given in their paper, of using it to find the effect of changing the air flow through the firebox of a steam turbine, suggests it is well suited to training of operators of systems, for example.

It is felt that these differences of aim between compositional modelling and the other approaches, all of which seem to be aimed more at using model based reasoning for design of systems, means that this approach can be considered more peripheral to the aim of this research.

4.2.2 Discussion of the model sets

The most apparent difference between the various sets of models illustrated above is the idea that they can be grouped by whether they are associated with a behavioural or functional approach. This seems to confirm the idea that there are two contrasting approaches. It is worth noting, however, that most of the reasoners do have the same aim, to generate a behavioural model of the subject system.

In most of the approaches illustrated, functional models are used as part of the simulation process, in that functional knowledge of components is used to derive the behaviour of the system as a whole. The exception is the approach illustrated in figure 4.2, (Price, 1998), where the simulation is purely based on knowledge of the system's structure and components' behaviours and the functional models are used to interpret this behaviour in terms of the system's purpose. This is the aim of the Functional Interpretation Language that is the result of the present work. The similarity of the model sets used for simulation in figure 4.2 and the

set in figure 4.3 suggest that a similar approach to interpretation of their simulation would be possible, using functional labelling or the Functional Interpretation Language. This further suggests that the present work, building as it does on the functional labelling approach, is not dependent on any specific approach to simulation. However, its use for interpretation of simulation does perhaps imply a behaviour based (as opposed to a function based) simulation.

In general the approaches illustrated all use a component centred ontology, the main exception being that of (Falkenhainer & Forbus, 1991), which is processed based. Therefore all uses of functional models seem to be associated with the component centred ontology. This is a natural, intuitive approach to design analysis, as the design of a system can be seen as the specification and connection of the components used in the system. That being the case, the present research was also centred on that ontology.

Having explored the background to the use of functional modelling as the basis for interpretation of model based simulation, it is time to consider the Functional Interpretation Language itself.

Chapter 5

Representation of function

This chapter will introduce and discuss the nature of the proposed language for the representation of function, in the light of the definition proposed in the previous chapter,

> an object O has a function F if it achieves an intended goal by virtue of some external trigger T resulting in the achievement of an external effect E.

This can be less formally described as a representation of function in terms of the triggers and effects of a system that result in its meeting some intended purpose, and as such in terms of how a device fulfils its purpose when viewed externally. The discussion will concentrate on the use of functional representation for the interpretation of the results of simulation, to drive the automated production of design analysis reports. The usefulness of the proposed functional representation for other tasks (such as functional reasoning and functional refinement of a proposed design) will be introduced briefly here but will be discussed at greater length in Chapter 11.

The definition of function as a relation between trigger, effect and purpose leads to the need for a representation of function to incorporate the three elements:-

- The trigger of the function can be thought of as the preconditions for the correct achievement of the function. This is a Boolean expression which, if it resolves to true, means the function's effects are expected. This allows several input states of the system to be identified and combined appropriately, using Boolean operators.

- The effect amounts to the post-conditions of the function. This is also a Boolean expression, so (loosely) the function is achieved when the post-conditions resolve to true. The use of Boolean operators allows cases where a function depends on several effects to be described.

- A representation of the purpose the function is to fulfil. A possible approach is to have a distinct teleological description that is referred to by the functional description, as being achieved by the function. This is discussed further below.

The trigger and effect between them provide a link to the behavioural model of the system, being concerned with inputs and outputs respectively. These three elements of a functional representation will be discussed in turn, following a discussion of the requirements of a functional representation and of the use of logic in recognising achievement of a system function.

As elements of the Functional Interpretation Language are introduced in this chapter, it is an opportune point to introduce the convention that elements of the language and quotations of functional descriptions are highlighted using a `typewriter typeface` and in addition keywords of the language, including the conventional logical relations, are in `CAPITAL LETTERS`.

5.1 Requirements of a functional representation

Before discussing the proposed representation of function, it seems worth briefly summarising requirements for a useful representation, specifically in the light of its use in interpretation of a simulation of an engineered system. The important requirements of a functional language are that it should be:-

1. Capable of recognising whether the purpose is being fulfilled. This naturally implies a mapping to some description of purpose. This is consistent with the reference to the idea of a function fulfilling a purpose in definition of function given earlier. This also requires a mapping to the simulated behaviour. What is required is some way of recognising whether or not the simulated behaviour fulfils an intended purpose of the system, so the behaviour is considered in terms of the truth or falsehood of the achievement of a function, and in turn the fulfilling of the intended purpose. This follows the functional labelling approach of (Price, 1998) and a logical basis for recognition of function that allows the development of recognisers for the achievement or otherwise of some function forms the subject of the next section.

2. Applicable to static or dynamic functions. This is one of the three desiderata for function in (Chandrasekaran & Josephson, 1996).

3. Applicable to physical and abstract objects. This is also one of the desiderata for function in (Chandrasekaran & Josephson, 1996) and is clearly significant if we are to use the representation of function in analysis of software based systems. The use of the proposed representation for interpretation of software based systems will be discussed later.

4. Independent of the system, specifically of the representation of its structure and behaviour. By this is meant that it should be possible to construct a functional description or specification of the intended system with no explicit reference to the details of the system itself. This increases the language's usefulness for functional refinement of a design and also improves the reusability of the functional model. One aspect of this is that the functional representation might be independent of the domain(s) of the internal workings of the subject system. For example, the purposes of gas and electric hobs are identical and at an admittedly rather high level of abstraction they both consume energy from some external source and convert that energy into heat. Therefore, at this abstract level, a similar functional description might be used for either. This independence from the system is important if the functional language is to be used to support the design of a system by functional refinement as the functional description will precede the specification of the details of the system. It also allows the functional language to be used for specification of detailed requirements of the planned system. The idea that the functional model is independent of the target system differs from the approach to functional labelling in (Price, 1998), where functions were attached to existing behaviours of significant components.

5. Capable of specifying expected behaviour to an arbitrary level of precision. What is meant by this that the functional model can be developed as far as is necessary to specify the intended (external) behaviour of the system. It will be appreciated that there is a likely conflict between the detail with which a functional model specifies the intended behaviour of a system and the model's reusability. For example, if the functional model of a gas hob specifies that to light a ring, the knob should be turned to some appropriate setting and then pushed in to trigger ignition of the resultant gas flow, then this description is not useful for an electric hob. This is valuable for functional specification of a system as the intended behaviour can be specified without reference to the implementation. One specific area where this is

valuable is if a functional specification is used to clarify the requirements of a system early in the design process. For example rather than simply stating that it is required to dip a car's headlamps the functional model could specify how this is to be achieved in terms of the user interactions, such as specifying that whenever the headlamps are first switched on they are dipped and that the dip switch itself is pulled and released to both dip the headlamps and return them to main beam, as opposed to simply having two positions of the dip switch. Later in the design process, the candidate design can then be verified against these requirements by simulation and interpretation of the results, using the specified functional description.

It will be seen that two of the three desiderata for a functional representation in (Chandrasekaran & Josephson, 1996) have been included. As the present work is concerned with functional representation in design analysis of engineered systems, the third, that a functional representation should be usable for both natural and man made systems, is not relevant. How the Functional Interpretation Language meets these requirements will discussed later in this chapter. In addition, there are the requirements identified in the introduction. These are that the language be capable of describing

- Systems where a function might be partially achieved.
- Systems whose functionality depends on intermittent or sequential effects.
- Systems where there is a danger of the required effects being achieved in an untimely manner (typically after an excessive delay since the function was triggered).
- Systems with subsidiary functions whose achievement depends on the state of some other system function.

How the proposed language approaches these requirements forms the content of Chapters 6 to 10.

As the first of these requirements is that the functional model can be used to establish whether or not a function has been achieved, the main rôle of the functional representation is to abstract the behavioural state of the system in these terms. The basis for this is described in the following section.

5.2 Logic and functional description

If a representation of function is to be used to interpret the results of simulation, in other words to express the system's behaviour in terms of its purpose, then some method of recognising the achievement (or otherwise) of the function is required, along with a representation of (or mapping to) the purpose the function is to fulfil. What is required is a mapping between the behavioural state of the system and the functional model, such that a behaviour can be identified as achieving the function or not, and through this whether or not it succeeds in achieving the system's purpose. To return to the simple torch example in Section 4.1.5.1, the simulation of the correctly working (and correctly designed) torch might, assuming a domain based electrical simulator, show that when the switch is closed, there is current flowing through the lamp (bulb). Informally, the definition of function given in Section 4.1.5.1 can be stated as "that relation between input (trigger) and output (effect) that achieves some purpose" so if we identify the purpose as being to "light the dark", the trigger with the switch being closed and the effect as current flowing through the bulb, then that behavioural state can be identified as fulfilling the torch's purpose. However, if a wire comes adrift, so it no longer makes the required connection, then switching on the torch no longer triggers the required effect and the torch no longer fulfils its purpose. Incidentally, in this case it seems safe representing the lamp as shining whenever there is current flowing if any failure of the bulb (such as the filament breaking) that prevents it lighting also prevents current from flowing. Such a condition might well be associated with some specific state in the component's behavioural model. For example a lamp might have a qualitative behavioural description along the lines of `if filament.i = ACTIVE then state is LIT` and that state might be used to map to the functional model of the torch. In this case the use of state is arguably an unnecessary complication, however. The relation between the simulation (and so the behavioural models) and interpretation (and so the functional model) is discussed in more detail in Chapter 11. Where a behavioural description is used in illustration of an example in the present thesis, it will be done (implicitly) in terms of state.

This approach to functional modelling follows (Price, 1998), with the change that the proposed representation of function includes the trigger as well as the effect. This is necessary for design verification and indeed is added to the functional model when the tool developed as a result of earlier work in Aberystwyth (*AutoSteve*) is used for SCA, see (Price *et al.*, 1996a). This approach has the advantage that a functional model need only be associated with the system as a whole and it need only be concerned with those components that form the "interface" between

the system and its environment, typically the controlling components (such as switches) and the effectors, simplifying the functional modelling. There is no need to add functional models for purely internal components (such as connections in the case of the torch) as the simulation does not require them and the system's functional model is stated in terms of its interfaces with the system's environment, any significant effects of malfunctions in internal components will show at these interfaces. This might require additional behavioural models that describe failure mode behaviours of components, at least for failure analysis.

The trigger and effect of a function are treated as Boolean expressions that between them serve to define the state of the function. The trigger can be seen as the precondition of the function, showing when the function's effect is expected, and the effect as its post-condition. The present representation of function differs from most other representations by its inclusion of the trigger that leads to the expected effect (output). The trigger and effect expressions are mapped to appropriate properties of the system's structural and behavioural model. This is discussed later. This has the advantage for the present work that the representation of function is intrinsically capable of showing the unexpected achievement of the effect, which does seem to be a potential weakness of the use of function. Naturally, if the trigger is false and the effect true, then the effect is achieved unexpectedly, unless, of course, the same effect can result from some other trigger, as part of some other function. This leads to the idea that there are actually four states that might be associated with the achievement of a function, as shown in Table 5.1. The table shows that if the trigger (precondition) is false, the function

trigger	effect	function state
False	False	**Inoperative (In)**
True	False	**Failed (Fa)**
False	True	**Unexpected (Un)**
True	True	**Achieved (Ac)**

Table 5.1: Achievement of function using trigger and effect.

should not be achieved, so the effect (post-condition) should also be false. If the effect resolves to true, there is something wrong. If the trigger resolves to true then the function should be achieved and clearly if the effect is false in this case then the function has failed. Therefore a function can fail in two ways — either by not being achieved when expected (referred to as failed, Fa) or the effect being achieved when not expected (achieved unexpectedly or unexpected, Un, in the Functional Interpretation Language). It will be seen that the trigger and effect

sharing a common value (whether true or false), is consistent with correct behaviour of the system. Either the function is correctly achieved or not intended, which case has been labelled inoperative (In). Despite this pairing off of values as being consistent with correct behaviour of the system (whenever the truths of the trigger and effect expression agree) or not, there remains a need to distinguish between the two cases of inconsistent behaviour(failed and unexpected). This is because the consequences of these two cases differ. As a function's state is consistent with the expected behaviour of the system if the trigger and effect have the same truth value, this consistency could be identified using the Boolean expression NOT XOR. However, this would lose the distinction between the two cases.

These functional states are used as keywords in the language so they can be used in specifying the triggering of some function that depends on the state of some other system function. This is discussed in Chapter 10.

The four function states, inoperative, achieved, failed and unexpected are defined in terms of the truth of the trigger and effect expressions, so for example a function having failed means that is trigger is true and its effect is false or in practice that the trigger has not resulted in the expected effect. The abbreviations as given in parentheses in the table are those used elsewhere in the thesis. Failed has been abbreviated as 'Fa' rather than 'F' to avoid confusion with false and the other three function states given similar two letter abbreviations. If we let f be some function that depends on a trigger expression t and an effect expression e, we can restate table 5.1 as four rules to define the four function states, inoperative, In(f), failed, Fa(f), unexpected, Un(f) and achieved Ac(f). The function f is said to be inoperative if neither the trigger not effect are present.

$$\text{In}(f) \Leftrightarrow \neg t \wedge \neg e \tag{5.1}$$

If the trigger is present but the effect absent, function f is failed.

$$\text{Fa}(f) \Leftrightarrow t \wedge \neg e \tag{5.2}$$

The effect being present without the trigger means f is unexpected.

$$\text{Un}(f) \Leftrightarrow \neg t \wedge e \tag{5.3}$$

If both trigger and effect are present the function f is achieved.

$$\text{Ac}(f) \Leftrightarrow t \wedge e \tag{5.4}$$

All the formal definitions of the rules used in the Functional Interpretation Lan-

guage are summarised in Appendix A on page 275.

In addition to these four functional states, it is sometimes useful to define the state of a function purely in terms of one or other of its trigger or effect. A function f is said to be triggered, $\mathrm{Tr}(f)$, if its trigger t is true regardless of the truth of its effect e.

$$\mathrm{Tr}(f) \Leftrightarrow t \Leftrightarrow t \wedge (e \vee \neg e) \tag{5.5}$$

Likewise, f is said to be effective, $\mathrm{Ef}(f)$, if its effect e is true regardless of its trigger t.

$$\mathrm{Ef}(f) \Leftrightarrow e \Leftrightarrow e \wedge (t \vee \neg t) \tag{5.6}$$

This allows the four function states inoperative, failed, unexpected and achieved to be defined in terms of these function states, by replacing the trigger and effect with the triggered and effective function states.

$$\mathrm{In}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \neg\mathrm{Ef}(f) \tag{5.7}$$

$$\mathrm{Fa}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \neg\mathrm{Ef}(f) \tag{5.8}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \mathrm{Ef}(f) \tag{5.9}$$

$$\mathrm{Ac}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \mathrm{Ef}(f) \tag{5.10}$$

It will be seen that these are identical to the four rules (5.1 to 5.4) given earlier. These relations are useful in the text and also for describing dependencies between functions, as discussed in Chapter 10. All of these relations can be traced back to the truth of the triggers and / or effects, and as such there is no ambiguity.

The operator `TRIGGERS` is used to represent the relationship between the trigger and effect, table 5.1 can be regarded as the truth table for this operator. The trigger and effect between them serve to show the state of achievement of the function, so in combination they serve as the recogniser for the function. The expression that precedes `TRIGGERS` describes the function's trigger and the keyword is followed by the required effect. For example, the recogniser for the functional description for a torch might read like this.

```
switch_on TRIGGERS lamp_lit
```

Here `switch_on` is a label that represents the trigger and `lamp_lit` the effect. These labels are then attached to appropriate properties of component behavioural description. This is described in Section 5.6.

There are arguably objections to the idea that the function is achieved unexpectedly if the effect is true and the trigger false. Pedantically, if the function is

to be defined as being dependent on both trigger and effect, then it should not be regarded as being achieved if either are false. This does lead to an interesting point, however, which is that the consequences of unexpected achievement of a function's effect do not, typically, relate to the purpose, while the consequences of failure of the expected effects do. To re-use the headlamps example, if the lamps do not light when expected the goal of lighting the road ahead is not achieved but if they are achieved unexpectedly, the consequences are dazzling of oncoming road users, unnecessary drain on the battery (if the engine is not running), and, of course, the road ahead might not be lit by the lamps at all, if it is daylight. There is therefore some asymmetry between the consequences of failure of a function and the unexpected achievement of its effect. This will be considered further in the sections discussing the effect of a function and the purpose to which the function relates.

As the trigger and effect are Boolean expressions, the Boolean operators can be used to specify the trigger and effect in cases where the achievement of a function depends on required combinations of triggers and effects, such as a combination of switch positions and / or several outputs being required. This is the simplest approach to decomposition of a system function and is discussed alongside the other possible approaches in the following chapter.

It appears to be the case that this four valued table applies less well to component function, as this will have a closer correspondence with its behaviour. Specifically it is unlikely that a component function's effect will be achieved unexpectedly as in this case the trigger will generally be the actual cause of the effect, instead of being merely one end of a causal network that leads to the effect. For example, a lamp's (that is a bulb's) function might be "light room achieved when filament current is active triggers shining". In this case the functional description adds little to the behavioural description as the behaviour of a lamp can be described in the same way, except for the reference to purpose, which arguably is more a property of the system the lamp is a part of, though it does (implicitly) suggest that the lamp should be in a room! As the proposed use of the functional representation is the interpretation of a system's behaviour with reference to its purpose, this is not seen as an undue difficulty. It is out of concern for the complexity of causality in a system that the term trigger has been preferred to cause. This is not to rule out the use of this representation of function for component functions, a lamp's function could be represented in terms of its being triggered by the presence of a current triggering the effect of light delivered to the rest of the system. However, it might be felt that the express description of purpose is superfluous at this level, unless functional models are used in simulation and a description of purpose is

useful in guiding the structural design of a system. For example, if a functional decomposition of some system includes the knowledge that a wire's purpose is to conduct electricity to some specified effector, this guides the structural design by implying that the wire be on a path to that effector.

Having shown that the state of achievement of a system function can be recognised in terms of the truth of pre and post conditions (its trigger and effect) the nature of the trigger and effect can now be discussed. In the simplest case, each need be no more than a label which acts as a hook to which some property of the simulated behavioural model can be attached.

5.3 The trigger

The trigger can be thought of as the required precondition for the function to be expected. If the trigger resolves to false, then the trigger's effects are unexpected and its purpose can remain unfulfilled. It might, of course, be the case that a given effect might be associated with more than one function, and so with more than one trigger. A possible example might be an industrial monitoring system where a hooter sounds to draw attention to any one of a number of failures in the process being monitored, which failure being distinguished by a different warning lamp. In this case, of course, the hooter sounding is not an unexpected effect of one such warning function if another process failure has triggered it.

Another view of the trigger is that it describes an interface with which a user (either human or some other encompassing system) interacts with the system being designed. It can be thought of the controlling condition that calls on the function. A switch is a typical example of such an interface. When specifying the trigger for a system function, any power supply will generally not be modelled as part of the trigger. This is simply because the power is either part of the system (such as a battery) or beyond the scope of the system design, such as the public electricity or gas supply, the testing of which is not really part of the design process for a specific mains powered device. An exception to this case is where the failure of the external power supply should result in some specific behaviour of the system, such as the starting of a backup power supply to allow the system to be shut down correctly. Even in these cases, the power supply need not be treated as part of the trigger of the main function, instead there will typically be a backup function triggered by the failure. This is discussed in Chapter 10.

The idea that the trigger is concerned with control of the system rather than driving it is one area where system level function typically differs from component

function. In the case of an effector component of a dynamic system, the input that triggers its function will be of energy, the task of the rest of the system being to deliver that energy and additionally control its delivery.

As has already been noted, a function might depend on several triggers which can be related using Boolean operators. It is quite possible that any of the three binary Boolean operators might be used in describing a trigger. The headlamp example mentioned above includes a trigger that combines two necessary components, combined using `AND`. While this is, perhaps the most usual operator, there are cases where the other binary operators (`OR` and `XOR`) might be used. An external light, for example, might be switched on either manually or by an automatic device that senses the ambient light level and switches on the external light if the level drops below some threshold. This can be described using `OR`:

```
light_switch_on OR light_level_low
TRIGGERS
outside_light_on
```

Here, of course, the simulation will need some way of modelling the idea that the ambient light is low. `XOR` is useful for describing the typical dual switched light in a stairwell, for example, where if one or other of the two switches is in the down position, the lamp should be lit:

```
ground_floor_switch_on XOR first_floor_switch_on
TRIGGERS
stairwell_light_on
```

In both these cases, of course, the effect and purpose associated with each trigger are the same. The use of the Boolean operators is discussed in greater detail in the following chapter, which discusses decomposition of function.

Having suggested that one requirement of a functional language is that it applies to static as well as dynamic systems, following (Chandrasekaran & Josephson, 1996), then a discussion of how the idea of a trigger might work in a static system seems necessary. The idea that a function is achieved by an effect resulting from a trigger does, it is appreciated, appear more consistent with dynamic systems. However there seems no difficulty in using the approach to describe a static function, such as support. For example, a bridge might be required to support a vehicle of some specified gross weight. This might be expressed as follows.

```
loading NOT greater than max weight TRIGGERS NOT collapse
```

The maximum weight is that intended (plus, presumably, some safety margin) and the effect of not collapsing might be defined in terms of a maximum deflection of the centre of the bridge. These can be specified when associating the functional description of the bridge to the simulation or could be specified in the functional description itself.

```
loading <= than 25 tonnes TRIGGERS deflection < 10 millimetres
```

On the one hand this seems a reasonable part of the functional specification of a bridge (though the figures might not be realistic!), but it does raise questions as to how best to associate the functional description with the simulation. An approach to associating simulations and functional models is discussed later.

5.4 The effect

At its simplest the effect in a functional description is, like the trigger, simply a label to which a property of the system's behaviour can be attached. The nature of this property is independent of the functional description itself. It might be an output, as will be the case if the system is a software module, or a goal state, such as a centrally locked car's doors all being locked. The effect does, of course need to be linked to some recognisable feature of the system's behavioural model. It will typically be some property associated with an effector component. It should certainly be some component whose effect on the system's environment is apparent.

While the use of logical operators is as applicable to effects as it is to triggers, in practice `OR` and `XOR` will seldom (if ever) be used to combine effects. This is because it is extremely unusual for a system to have a trigger that triggers, non-deterministically, one or both of two possible effects, such that either is an acceptable way of fulfilling the same purpose. Indeed, if this seems to be the case, then typically the trigger should be better defined. In the case of a software module, for example, an input might result in one of two outputs but if there is no need to distinguish between the outputs (they both fulfil the same purpose) then what is the purpose of the module? It is, perhaps, not impossible that for some analyses, all that is required is to know that the module handles the input concerned without crashing, in which case, arguably, the different outputs might be immaterial. This seems unlikely but should not be ruled out. This, combined with the possibility of wanting to describe a non-deterministic system, suggests

that the use of OR and XOR should be allowed in combining effects. One possible approach to this case might be to have a tool that implements the functional language warn the user about the use of these operators in this situation.

It does seem worth noting that where a description of function is used to interpret a simulation, then the effect on which the function depends does need to be capable of representation in a simulation. This does restrict its use for cases where a function is aesthetic and / or subjective, such as the function (strictly, the purpose) of placing a potted plant in the room is to improve the room's ambience. The anti-corrosive properties of paint could be represented, while the aesthetic properties could not.

Having considered the two components of a functional description's recogniser, we can now consider the representation of the purpose the function is to fulfil. Note that from hereon a single function, associated with the achievement of some purpose of a system will be referred to as a "functional description", the term "functional model" being reserved for the set of functional descriptions associated with some system. In the case of a simple system like the torch, therefore, the functional model might contain but one functional description. However, for many systems, the system's functional model will include several functional descriptions. For example, a car exterior lighting system's functional model will include functional descriptions for headlamps (dipped and main beam), tail lamps, brake lights and so on.

5.5 Representation of purpose

At its simplest, in principle, a description of purpose need consist of no more than a textual description of the purpose, so the purpose of a torch might be no more than the entry "help the user see in the dark". This could, of course easily be incorporated a functional description. The keyword `ACHIEVES` is used to indicate the purpose associated with a function, so the torch functional description can be expanded to this.

```
FUNCTION torch
  ACHIEVES "help the user see in the dark"
  BY
    switch_on TRIGGERS lamp_lit
```

The keyword `FUNCTION` labels the function itself and `BY` indicates the recogniser. However, for design analysis there is more interest in unexpected results (whether

failure to achieve the required effect or unexpected achievement of an unwanted effect) so that the functional description in (Price, 1998) includes a description of the consequences, both of failure and unexpected achievement of a function. As the consequences of failure of the function (the expected effects not being achieved) will typically relate to the purpose of the function (so if the torch fails to light, the user is not helped to see in the dark) these can neatly be added to the description of purpose. Similarly, the values for severity and detection, used to arrive at the failure's risk priority number (see Section 2.1.1.5 on page 24), can also be added to the description of purpose. The value for occurrence is associated with the likelihood of the component failure that gives rise to the failure of the function, so is included in the component's behavioural model. It is not part of the functional description. The torch's functional description might now be further expanded.

```
FUNCTION torch
  ACHIEVES "help the user see in the dark"
    FAILURE_CONSEQUENCE "user not helped to see"
    SEVERITY 5
    DETECTION 1
  BY
    switch_on TRIGGERS lamp_lit
```

Separating the description of purpose (teleological model) from the functional description itself simplifies the functional description. It is also consistent with the idea that function is concerned with how a purpose is fulfilled rather than the purpose itself. Therefore the torch's functional description might look like this.

```
FUNCTION torch
  ACHIEVES see_in_dark
  BY
    switch_on TRIGGERS lamp_lit
```

Here `see_in_dark` is a reference to a separate teleological model.

```
PURPOSE see_in_dark
  DESCRIPTION "help the user see in the dark"
  FAILURE_CONSEQUENCE "user not helped to see"
    SEVERITY 5
    DETECTION 2
```

This has the practical advantage of assisting in reuse of existing information. For example, the purpose, consequences of failure and severity value of a stop lamp

system on any road vehicle are similar and such information could well be reused by functional models that associate the function with the lighting of one lamp (on a motorbike), two (on a car) or three (if there is an extra repeater fitted). The alternative is, of course, to use the same functional description in each case, but this cannot be done without changes to the composition of the effect, at the very least, so reusing only the teleological description might be simpler and more convenient in this case. The idea that a purpose is external to the system (and therefore generally to the simulated world) makes a formal description more difficult and arguably unnecessary. One of the purposes of the functional language is therefore to provide a connection between the informal teleological description and the formal representation of behaviour associated with the reasoner.

While the consequences of failure of a function will be associated with the purpose the function is associated with, the consequences of unexpected achievement of the effects will frequently not be. For example, the consequences of a torch lamp staying on when switched off are concerned with draining the battery rather than seeing in the dark. Therefore these consequences typically will not be included in the teleological model but instead will be associated with the link between the functional description and the simulation, as will be discussed later. However, there are occasions where unexpected achievement of a function's effects do relate to the system failing to fulfil its purpose. For example, if a car's stop lights stay lit when the brake pedal is not being pressed, the driver of a following car is not shown that the car is stopping. Therefore, a teleological model can, optionally, include a similar set of consequences, and values for severity and detection for unexpected achievement of a function's effects. The idea that the consequences of unexpected achievement of a function's effects are attached to the link between the functional model and the system's behavioural representation is discussed in the following section, on the relationship between those two models. Note that the inclusion of the values for severity and detection will depend on the use to which the design analysis is being put, and they need not be present. In this thesis, their use is assumed, partly because their presence is of help in clarifying features of the Functional Interpretation Language.

5.6 Relationship with the simulation

The inclusion of the trigger and the use of labels to represent the trigger and effect of the function allow the functional description to be built independently of the target system. This has the advantage that the language can be used for tasks in which the functional model is built before the system itself, such as supporting

the functional refinement of a design and capturing functional requirements of a possible system. It should also encourage reuse of functional models.

For interpretation of a simulation, we naturally require a mapping between the system's behavioural model and the functional model. The approach taken here is to link a complete functional description with appropriate properties of the system's behavioural model. This does follow (Price, 1998) in that the models are actually those of significant components, that is input components (such as switches) and effectors. These behavioural models are labelled as implementing a trigger or effect of a function, from some existing functional description. For example, the component models of the torch will include a switch and a lamp (bulb). The switch might be modelled as having two positions, open and closed. On attaching the model of the torch system to the torch functional description, the switch being closed will implement the trigger of the torch function. The model of the torch will then have the following attached on associating the system with an appropriate functional model.

```
switch.position = closed IMPLEMENTS torch.switch_on
```

Here, `torch` is, of course, a reference to a functional description and `switch_on` a reference to that model's trigger. Naturally such an attachment will use components that control inputs to the system being simulated

Implementing effects is a little more complex. An effect might be an output (such as light) or a goal state, such as a car's central locking system having all the doors locked. It also requires the consequence of unexpected achievement of the effect adding. If an effect is achieved unexpectedly (that is, when it is not triggered) this failure of the system will have its own consequences, distinct from those associated with failure to achieve an expected effect (that is, failure of the function). These consequences will, like the consequence of failure of a function, consist of a textual description of the consequences and (depending on the use to which the language is being put) values for severity and detection. Attaching these consequence to the association between system and functional description maintains the independence of the functional description from its target system and so encourages reuse of the functional description itself. The IMPLEMENTS connection for the torch effect is as follows

```
lamp.state = lit IMPLEMENTS torch.lamp_lit
  UNEXPECTED_CONSEQUENCES "Drain on battery"
  SEVERITY 3
  DETECTION 2
```

The lamp has a (possibly redundant) state description. All that is required is that the property of the system that implements an effect can be recognised by the simulation. In this case the state description could be dispensed with and a (qualitative) current used instead, as shown here.

```
lamp.current = active IMPLEMENTS torch.lamp_lit
  UNEXPECTED_CONSEQUENCES "Drain on battery"
  SEVERITY 3
  DETECTION 2
```

This does assume that there is no failure of the bulb that allows current to flow through it without it lighting. In general, where such associations are shown in this thesis, they will be shown using state, for the sake of consistency. The nature of the relationship between the functional model, its target system and the simulation is discussed in more detail in Chapter 11.

One aspect of the links between a functional description and its associated system is how this affects the precision of the functional description. Should the functional description itself specify the possible triggers of a function, or should this be left to the association between function and system? To illustrate, a torch might have a sprung push-to-make switch to switch on the torch when it is only required for a short flash, as well as the two position switch. There are three possible approaches to representing this case. The simple functional description given earlier could be used, with the alternative trigger specified in the association.

```
slider_switch.position = closed OR push_button.position = pressed
  IMPLEMENTS torch.switch_on
```

This simplification of the functional description itself does, of course, encourage its reuse, but at the expense of precision in the description of the expected functionality. One advantage of the use of labels for representing the trigger (and effect) is that this precision can, if required, be incorporated into the functional description itself.

```
FUNCTION torch
  ACHIEVES see_in_dark
  BY
    switch_on OR flash_on TRIGGERS lamp_lit
```

This shows that an alternative way of switching on the torch is required, with identical effects (and purpose). It will require two associations with the system.

```
slider_switch.position = closed IMPLEMENTS torch.switch_on

push_button.position = pressed IMPLEMENTS torch.flash_on
```

An alternative (as in this case) might be to associate the `flash_on` function with its own purpose (signalling, perhaps), with the two functions sharing an effect and so have two separate functional descriptions, one for each intended use of the torch. There are cases where one might or might not wish to associate an effect with alternative triggers, but where there is no question of a distinct purpose the option of using different functions is arguably not appropriate, though the two functions could be associated with one description of purpose. These alternative approaches allow the intended functionality to be specified to an arbitrary level of precision, subject to an inevitable "trade off" between precision and reusability of the functional description. This was one of the requirements of a functional language listed earlier in this chapter. The choice of how much precision to use will be at least partly dependent on how the functional model is used. If a system's functional model is to be used simply for interpretation of simulation, the simplest model is likely to be adequate, but if functional modelling is used to guide the design process then the more elaborate model might be more appropriate as ensuring all intended behaviours are allowed for in the physical design. For example, the functional model above shows that the design of the torch is to allow for it to be flashed briefly on and off.

There is less need for the use of logical operators to combine implementations of a function's effects than triggers. The effects are typically all going to be required for achievement of the function rather than being alternatives, as might well be the case for triggers. Also, of course, the effects are also what actually achieves the function, so a correct functional description will specify them with adequate precision. For example, the effect clause in a car headlamps main beam function should specifically refer to the effect of both right and left headlamps. This association with the effects of a system might rely on an indirect indicator, so in the case of the headlamps system, the presence of current through the lamps can be taken as indirect evidence of the output of light (the real effect) given that there is no failure of a lamp that allows current to flow through the lamp without it giving off light.

Attaching the consequences of the unexpected achievement of an effect to the mapping between system and function has the advantage that the model includes the unexpected consequence of any effect, even where it does not amount to achieving a function. In the approach taken by (Price, 1998), the consequences of both failure and unexpected achievement were associated with a function, with the

result that the consequences of unexpected achievement of an effect were not reported if that effect did not itself amount to achieving a function. For example, a car headlamp main beam function might look like this.

```
FUNCTION main_beam
  FAILURE "Driver cannot see road ahead"
    SEVERITY 8
    DETECTION 2
  UNEXPECTED "oncoming drivers dazzled"
    SEVERITY 6
    DETECTION 2
  BY
    right_main_filament.current = active
    AND
    left_main_filament.current = active
```

Note that this is not an accurate rendition of the functional label approach in (Price, 1998), but is a plausible one, that carries the same meanings. There are two differences between this and the Functional Interpretation Language presented in this thesis that while appearing slight are nonetheless significant. The first of these is the lack of a trigger expression (which for FMEA is derived from the correct simulation). Its inclusion in the Functional Interpretation Language allows the state of some system function to be included in the trigger expression for some other function, as discussed in Chapter 10. The second and the use in functional labelling of existing properties of the system as effects that achieve the function. The Functional Interpretation Language's use of labels for the effect, which are later associated with the appropriate system properties, allows a functional description of a system to be built independently of the structural design.

It will be seen that as the function depends on both headlamps' main beam filaments being active, if one lamp is on, the function is not achieved. Therefore any fault that causes one lamp to stay lit the whole time (possibly a wire shorting to ground) does not achieve any function (correctly) but the consequence of the unexpected effect is lost, as it is associated with the achievement of the function. One headlamp being full beam is still capable of dazzling oncoming drivers but this is not reported. As implemented in the commercial tool (*AutoSteve*) there is a way of avoiding this problem, by associating each lamp with its own function, so the main beam function has two subfunctions, left main and right main. There are objections to this solution, in that it complicates the functional description and anyway as one headlamp on its own does not really fulfil any purpose (both are required to drive after dark) its effect does not really constitute a function. Attaching the unexpected consequences to the mapping between the part of the

system that implements a function's effect and the function avoids this difficulty as each effect than has its own consequences of unexpected achievement. In the Functional Interpretation Language presented here the same system's functional description might look like this.

```
FUNCTION main_beam
  ACHIEVES light_road_ahead
  BY
    lamp_switch_heads AND dip_switch_main
    TRIGGERS
    left_main_beam AND right_main_beam

left_main_filament.current = active IMPLEMENTS left_main_beam
  UNEXPECTED_CONSEQUENCE "oncoming drivers dazzled"
    SEVERITY 6
    DETECTION 2

right_main_filament.current = active IMPLEMENTS right_main_beam
  UNEXPECTED_CONSEQUENCE "oncoming drivers dazzled"
    SEVERITY 6
    DETECTION 2
```

The two identical unexpected consequences could, of course, be a single separate description, applied to both effects. In this case the consequence of either headlamp being lit on main beam unexpectedly is not lost and the achievement of the headlamps main beam function still depends on both lamps being lit correctly. There is still a rôle for subsidiary functions, discussed in the following two chapters.

Where a function depends on more than one effect, it is possible that if both or all of the effects are achieved unexpectedly, the consequences are different from the unexpected achievement of one. For example if one brake light stays on when the pedal is not pressed, it is wasteful and distracting to a following driver, but at least some warning that the vehicle is slowing is given by the other brake light working normally. If they both stay on, no warning is given. This is why a description of the consequences of a function's effects can be attached to a description of purpose. It is likely that where all a function's effects being achieved unexpectedly have different consequences from any one of them, this will result in the purpose not being fulfilled. It is still necessary in these cases to provide a set of unexpected consequences to each mapping between effect and behaviour, to capture the consequences of that effect alone being achieved unexpectedly. This is discussed further in the chapter on functional decomposition.

5.7 Functional description and report generation

Having discussed the relationship between the functional description and the simulation, allowing its interpretation, it is worth introducing how the functional description is used to generate a design analysis report. This report will include entries for all behaviours that result in unexpected functional states and the consequences of these unexpected function states.

The reporting of the state of a function can readily be described in terms of the state of the function, as described in Section 5.2 above. Let f be some device function associated some description of purpose and through that with consequences of failure of the function c_f. In addition, each effect is associated with the consequences of unexpected achievement of that effect. In the simplest case, the effect expression has a single element e with unexpected consequences u_e. The report will include an entry for each function that is in a state that is not consistent with expected behaviour of the system. Let us use the notation $\mathrm{R}(f)$ to mean "the resulting design analysis report includes f" and $\mathrm{R}(c_f)$ to mean the report includes the consequences fop failure of f. The report will include references to functions as follows.

$$\mathrm{R}(f) \text{ if } \mathrm{Fa}(f) \vee \mathrm{Un}(f) \tag{5.11}$$

So the report will include any function that is failed or unexpected. The report should also include a reference to the individual effect e that led to this inconsistent behaviour.

$$\mathrm{R}(e) \text{ if } (\mathrm{Tr}(f) \wedge \neg e) \vee (\neg \mathrm{Tr}(f) \wedge e) \tag{5.12}$$

So any effect that is absent when it is expected or present when it is not triggered will also be included in the resulting report. It should be noted that where an effect expression includes several individual elements (as in the headlamp example) each element is included following this rule, and therefore an element might be included in a design analysis report even though it does not result in a function being either failed or unexpected. For example, if one of the headlamps was lit unexpectedly, this does not amount to the function being unexpected (as the effect expression is false) but that headlamp being lit is still included in the report. The following chapter discusses this. The use of these rules is illustrated in the example that follows and forms the bulk of this section.

The report should also include the consequences of these inconsistent behaviours. As the consequences of failure of a function are associated with that function (through its description of purpose) the consequences of failure can be reported

when the function is failed.

$$\mathrm{R}(c_f) \text{ if } \mathrm{Fa}(f) \tag{5.13}$$

Because the consequences of unexpected achievement of an effect are associated with that effect's mapping to the system model, this can be reported like this.

$$\mathrm{R}(u_e) \text{ if } \neg\mathrm{Tr}(f) \wedge e \tag{5.14}$$

So the consequences of any effect are noted even if this does not amount to a function being unexpected where a function depends on more than one element in the effect expression. Note that while the preceding descriptions serve as formal rules for the simple example used in this section, they are simplifications of the rules used in the Functional Interpretation Langauge. In the case of 5.13, this is because a function might be an Operational Incomplete Function, described in Section 7.2, which has no associated description of purpose and therefore no consequences of failure. A function might optionally be associated with consequences of its being unexpected. This is discussed in Section 6.2.3. This entails additional conditions in the rule derived from 5.14.

These formal rules give a more complete foundation to what is essentially the same approach as is already used in the design analysis tool developed in earlier work, but it does seem worth describing its use as an introduction to provide background to the extensions of the expressiveness of the approach that the present Functional Interpretation Language supports. This will be done primarily using the simple torch example used earlier. It seems worth recapitulating the functional description and associated description of purpose together with the mappings between the system and the functional model.

```
FUNCTION torch
  ACHIEVES see_in_dark
  BY
    switch_on TRIGGERS lamp_lit

PURPOSE see_in_dark
  DESCRIPTION "Help the user see in the dark"
  FAILURE_CONSEQUENCE "User not helped to see"
    SEVERITY 5
    DETECTION 2

switch.position = closed IMPLEMENTS torch.switch_on

lamp.state = lit IMPLEMENTS torch.lamp_lit
  UNEXPECTED_CONSEQUENCE "Drain on battery"
  SEVERITY 3
```

```
DETECTION 2
```

The resulting report will typically be in a tabular format, with columns for failure cause and effect, consequences (possibly) and severity and detection. In generating a report in such a format, all that is required of the design analysis software is to look up the functional model and so establish which functions are achieved, which have failed (that is triggered but with the effect absent) and what, if any, effects are unexpected. If the torch is designed as in the little schematic in Figure 5.1 then if there are no component failures, the torch will work as expected. When the switch is closed, the lamp will light. For FMEA, the design analysis report will include

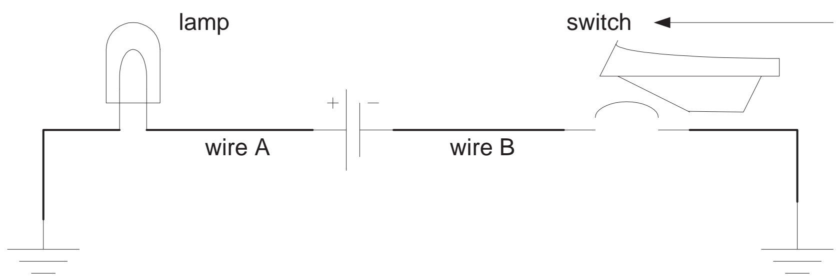


Figure 5.1: Circuit diagram for the torch example

the consequence of each failure of any component in the circuit. Here two failures are included for brevity, wire A breaking (going open circuit) and wire B shorting to ground. It will be seen that the result of running an electrical circuit simulation on this circuit with wire A broken will show that no current will flow through the lamp whether or not the switch is closed. The lamp's behavioural model will then show that it will not enter the lit state. The design analysis software can then examine the functional description, which shows that when the switch is closed, the function `torch` is triggered, but is not achieved because the lamp is not lit. Likewise, the electrical simulator will show that when wire B shorts to ground, current flows through the lamp (so it stays lit) whether or not the switch is closed, as the switch is bypassed by the short to ground. On examining the functional models, the software has shown that there are no functions that result in the lamp being lit with the switch open, so the effect is achieved unexpectedly. The functional models (and descriptions of purpose) already contain the text to be incorporated into the design analysis report, so this can readily be generated, and might resemble Table 5.2. In the first result, the failure effect field includes the name of the failed effect, consistently with rule 5.11 above and also includes the effect that caused the failure of the function, from rule 5.12. As the effect has failed, rule 5.13 has the report including the consequences of the failure, in

Failure effect	Cause	Consequence	Sev	Det
When switch_closed, function torch failed because expected effect lamp_lit was absent	wire A open circuit	User not helped to see	5	2
When switch_open, function torch was unexpected because unexpected effect lamp_lit present	wire B short to ground	Drain on battery	3	2

Table 5.2: Part of an FMEA report for the torch example.

the third column. The second result includes the unexpected function and the unexpected effect that causes the function to be unexpected, from rules 5.12 and 5.13 but in this case the consequences are those of the unexpected effect, from 5.14.

A complete report will, of course, have other columns, including one for the occurrence value (which is a property of the failure cause and so of the model of the component concerned), one for the RPN value and various columns that will be filled by hand, describing any action to be taken. Where generated text is quoted in this thesis, a simple textual format has been used, as it was felt this was easier to read. The textual form used for the first of the failures listed in the table would read

> Function see_in_dark not achieved because expected effect lamp_lit was absent. Consequences are: User not helped to see. Severity 5, detection 2.

These textual quotations from possible design analysis reports will generally be shortened to include only the text required to illustrate the point being made. In both cases it will be seen that most of the text is taken from an existing model, either the functional description, the description of purpose, or the mapping between system and function. Only simple linking words or phrases are added, such as the 'because' linking the function failure with the missing effect. It will be seen that the necessary text is easily generated and, indeed, this technique is in use from the earlier work. In this case, the labels for trigger and effect in the functional description have been used, but it will be appreciated that it would be straightforward for the software to use the mappings between system and functional models to generate text that includes the states of components in the system model. The only novel feature here is the use of the effect label to give some explanation of the failure of the function. This is clearly more interesting where a

function depends on several effects, as the missing effect(s) can be specified. For example, a report on the headlamp example might include "Function main_beam not achieved because expected effect left_main_beam was absent".

As the design analysis report will include repetitious entries of failure effects, even in this simple example any wire going open circuit will have the same effect, the use of functional descriptions and associated models means that such text is only written once, on creating the initial models. This both saves time, avoiding repetitious work, and reduces the danger of inconsistency between similar failures. For example, there is now no danger of the effects of wire B going open circuit being any different from those of wire A.

In Table 5.2 the different entries in the report are associated with different states of the torch's functional model. Wire A going open circuit caused the function to fail (trigger true, effect false) so the consequences are those associated with failure of the function. In contrast, wire B shorting to ground caused the unexpected achievement of the effect of the lamp being lit, so the consequences are those associated with that effect. This relationship between the state of the function (failed or unexpected) and the generated text can be generalised for a simple functional description is shown in Table 5.3. Here a general form of the generated text for

function F			generated text	consequences
t	e	state		
F	F	**I**	(no entry)	
T	F	**Fa**	Function F failed because expected effect e was absent	Fa(F)
F	T	**U**	Function F achieved unexpectedly because unexpected effect e was present	U(F) or U(e)
T	T	**A**	Function F achieved	

Table 5.3: States of a function and the resulting text.

an automatically generated report is associated with the state of a function. In the table, the consequences associated with any inappropriate result are treated implicitly as a set that consists of a textual description, a value for severity and one for detection. They are denoted by the abbreviated form of the functional state (failed or unexpected) associated with the name of the function or effect concerned, so Fa(F) means the consequences associated with the failure of function F. The alternatives for the consequences of unexpected achievement are, of course, because the function itself might or might not have a set of consequences of unexpected achievement associated with it. In such a simple case, it is unlikely that it would as they would, of course be much the same as the required ones of

achievement of the effect. The notation used in this table and other similar tables that appear in the following chapters is described in more detail in Appendix D on page 287. The table illustrates the straightforward relationship between a functional description and the generated text in a resulting design analysis report.

5.8 Notation used

Two notations are used from here on, a textual one that is consistent with the file format for a functional description and a graphical notation intended to illustrate the relationships between different elements of the functional description itself and between that description and other related parts of the system model.

The textual notation has, of course, already been used in the example functional description included above, and is also described in Appendix B. A brief summary of the notation and conventions governing its use in this text seems useful here, however, as it will be used in illustration throughout the remainder of the thesis. The headlamp main beam functional description used above is shown as follows.

```
FUNCTION main_beam
  ACHIEVES light_road_ahead
  BY
    lamp_switch_heads AND dip_switch_main
    TRIGGERS
    left_main_beam AND right_main_beam

PURPOSE light_road_ahead
  DESCRIPTION "Allows driver to see road ahead after dark."
  FAILURE_CONSEQUENCE "Driver cannot see after dark."
    SEVERITY 8
    DETECTION 3

light_switch.position = heads IMPLEMENTS main_beam.lamp_switch_heads

dip_switch.position = main IMPLEMENTS main_beam.dip_switch_main

left_main_filament.current = active IMPLEMENTS main_beam.left_main
  UNEXPECTED_CONSEQUENCE "oncoming drivers dazzled"
    SEVERITY 6
    DETECTION 2

right_main_filament.current = active IMPLEMENTS main_beam.right_main
  UNEXPECTED_CONSEQUENCE "oncoming drivers dazzled"
    SEVERITY 6
    DETECTION 2
```

The description of purpose and the mappings to the system model are not themselves part of the functional description itself. A blank line in the notation is used to separate parts of a description that are separate. These might be separate files, or possibly different entries in a database of functional representations. Such implementation issues are discussed in Chapter 11. The textual elements have been kept brief to save space and the arrangement of lines and indentation are intended to assist with reading the descriptions. Keywords of the language have been printed in block capitals to distinguish them from other elements of the functional models, all of which are lower case.

This notation will be expanded in the following chapters to include other parts of the language. There is a full listing of the language in Appendix B, on page 281.

5.9 Discussion

It seems worth recapitulating some of the general points this somewhat discursive chapter has raised, and further describing the relationships between the different models (or model fragments) used in the approach described above. This section will also briefly summarise how the language meets the requirements enumerated in Section 5.1.

The proposed approach to the use of this functional representation language for interpretation of simulation in terms of purpose follows the functional labelling approach of (Price, 1998) so that the trigger and effect are linked to the states (the behaviour) of the relevant components in the system, and are used to interpret the state of the simulation in terms of the achievement of the function, so they allow achievement of the function to be recognised, fulfilling the first of the requirements for a functional description.

The approach suggested in (Price, 1998) is that functional labels are attached to the relevant component descriptions, specifically their outputs. The approach described here reverses that in that the functional description is constructed independently of the system model, which is later attached to the functional model (using IMPLEMENTS). This change in the use of functional description is supported by the explicit inclusion of the trigger and by the use of labels to mark the expected trigger and effect associated with a function. The inclusion of the trigger increases both the range of design analyses that are supported by the language and also the range of systems that can be analysed. If the trigger is omitted, then the preconditions for a function are derived from a simulation of the system when working with no failures, which relies on the assumption that this is an accurate

reflection of the intended behaviour of the system. This is consistent with failure analysis, but not with design verification. Even for failure analysis, though, a system function that is triggered by failure of some other system function (such as a warning function) cannot be unambiguously derived from a failure free simulation. This is because the triggering conditions will ever be achieved in the correct simulation, and must be stated in terms of the state of the function on which the function concerned depends. There is some other triggering condition needed to trigger a fault mitigating function. This is discussed in Chapter 10.

The reversing of the earlier approach, so that system properties are attached to an existing functional description rather than the other way around, means that the functional model can be built independently of the target system, fulfilling the fourth of the requirements listed in Section 5.1. This allows the language to be used to support design activity as well as analysis. This independence from the system enables the required functionality to be described in as much detail as is necessary to specify functional aspects of the design. This was illustrated using the torch example earlier. In that case, the functional description in which the lit function was triggered by either a slider switch or by a push to make switch (for shorter flashes) can be used to specify that a way of flashing the torch on is a requirement. This could, of course, be done in more specific terms than in the example as given above, like this.

```
FUNCTION torch
  ACHIEVES see_in_dark
  BY
    slider_closed OR button_pressed TRIGGERS lamp_lit
```

This fulfils the fifth of the requirements listed in Section 5.1 and supports the use of the language for functional refinement of a design and enables its use as a vehicle for clarifying design requirements. The use of labels for triggers and effects also makes the functional description more self contained, which encourages reuse of existing models.

The use of a separate description of purpose is another difference between the present language and the earlier functional labelling approach. This further encourages reuse of existing models, as the description of purpose can be reused even in cases where systems that fulfil that purpose need different functional descriptions, such as the differences between a car's and motorbike's lighting systems. The relationships between the functional description itself and the other models used to map between a system's behaviour and its purpose are shown in Figure 5.2.

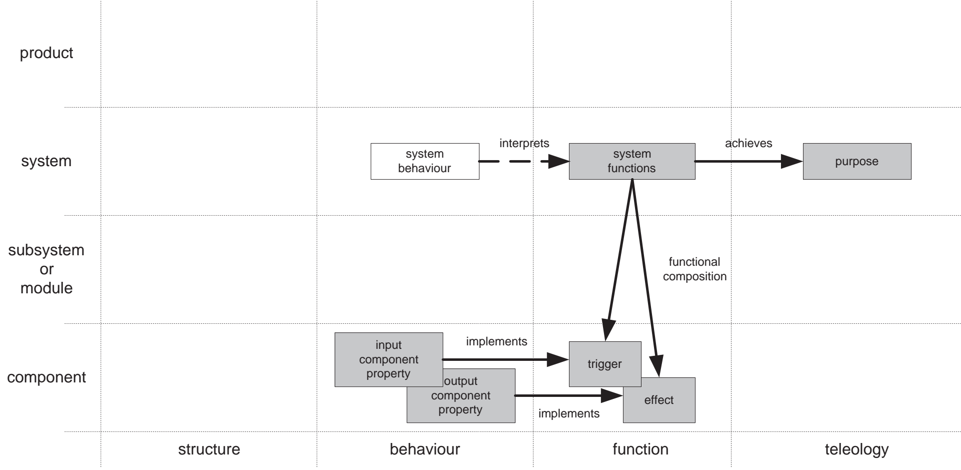


Figure 5.2: Model relationships using the proposed functional representation

The use of Boolean expressions uniting aspects of the required trigger and effect allow the functional language to be used to describe functions that depend on combinations of inputs and outputs of the system in a similar way to that proposed in (Snooke & Price, 1998). How the language manages different aspects of this is discussed in the next chapter.

The proposed language also has two other dimensions besides the hierarchy. These are new. The intention is that they will be introduced here and then discussed in greater detail in later chapters. These are

- The need to model intermittent and sequential behaviours and the ordering of state transitions.
- The need to allow modelling of cases where a function is achieved but not in a timely manner (typically late). This is increasingly important with the use of Carrier Sense Multiple Access / Collision Detection (CSMA/CD) networks to communicate between controllers and effectors within a system. This is because such networks cannot guarantee the prompt arrival of a message. The use of Controller Area Networks (CAN) in the automotive sector is an example.
- The need to model functions whose achievement should depend on the achievement or otherwise of some other function.

It can be argued that these provide additional functional relationships that are orthogonal to the straightforward logical hierarchy in the existing language. They add a temporal axis and a dependency axis that are both distinct from the hierarchical function / subfunction axis.

The approach proposed has the description of purpose that the function is to fulfil as a separate model (file) and also allows for elements of the system's behavioural description to be attached to parts of the functional description. The models used for the simple torch example used earlier are illustrated in Figure 5.3. This is

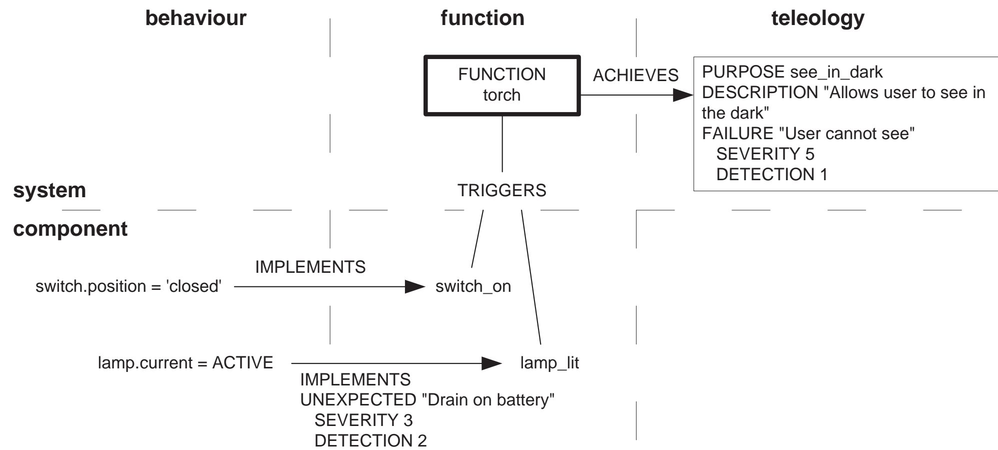


Figure 5.3: The relationships between the elements in the torch functional description

a more concrete rendition of the set of mode relationships shown in Figure 5.2 above. Here a thick walled box is used to denote a function, and arrows are used to indicate links between separate model elements. There is a full key to this and similar diagrams in Appendix C. While the figure shows a relatively large number of models (or model fragments), they are relatively simple and, as suggested above, reusable, so in use it will often be the case that a system can simply be associated with existing functional models or, failing that, existing descriptions of purpose. These can simply be retrieved from a library of models and model fragments. The functional model itself is simplified by only having to refer to inputs and outputs (or goal states) of the system as a whole. Those components whose effects are internal need no functional representation. This follows the functional labelling approach. Further simplification of the functional description is achieved by separating it from the description of purpose.

How the language can be used to describe the function of static systems (the second of the requirements listed in Section 5.1) was discussed in Section 5.3. It could be argued that the use of the functional language to interpret the results of a simulation is more applicable to dynamic systems, but it could be used to interpret a simulation in which the goal was for nothing to happen.

As a functional description is free of details of the target system any appropriate system property can be chosen as implementing the function's triggers or effects.

Effects will typically by associated with an output, such as of light in the torch example, or with a goal state such as all doors being locked in a central locking system. This supports the description of software components, either as part of a larger system or separately. For example, a networking subsystem (such as CAN) might have a functional description like this.

```
FUNCTION send_message
  ACHIEVES pass_information
  BY
    data_to_send TRIGGERS message_broadcast
```

Here, the arrival of data to send is labelled by `data_to_send` and the function is achieved if all nodes on the network receive the message, labelled with `message_broadcast`. Note that this model is simplified by the lack of any details of how the function is achieved, the function is achieved if the message gets through. For example the protocol is not specified, though the functional description could be refined so as to suggest requirements that might influence the choice of protocol.

Alternatively, an individual software component could also be associated with appropriate functional descriptions. This might be done at a variety of levels, such as an object in Object Oriented programming or an individual procedure (or method). The use of the functional description with software is discussed in more detail in Chapter 11. This is an important application of the third of the requirements listed in Section 5.1, that a language be applicable to abstract as well as physical systems.

The language as described in this chapter provides a basis for approaches to fulfilling the additional requirements listed in the Introduction. How the language does so is discussed in the following chapters.

While the use of Boolean expressions to represent the trigger and effect of a system function allows functions that depend on more than one condition for the trigger and effect to be described, this places limitations on the expressiveness of the language, especially in its use for functional refinement of a design. There are cases where a function is more accurately represented in terms of subsidiary functions rather than combinations of triggers and effects. The following chapter considers the different decompositions of function that might be used, and describes the language's support for these decompositions.

Chapter 6

Functional decomposition

In all but the simplest of systems, a system function will depend on more than one trigger and / or effect and the use of Boolean expressions to describe the relationship between elements in these complex triggers and effects has been introduced in the previous chapter. This chapter will discuss the decomposition of function in rather more detail, introducing and discussing the idea that a function might instead be composed of subsidiary functions with their own trigger, effect and purpose. This enables the Functional Interpretation Language to describe functions that have several elements each with its own distinct purpose that nonetheless contributes to the achievement of the main function.

The idea that a function can be thought of as being composed of either trigger and effect or subsidiary functions differs from the approach in (Snooke & Price, 1998). In that paper, a function was considered to be composed of subfunctions. For example, a car's headlamp main beam function would be composed of left main beam and right main beam subfunctions. This allowed the automatically generated design analysis report to describe a failure in more detail, so if the left headlamp failed then the report might state "function headlamps main beam failed because left main beam failed". The approach described here extends this increased expressiveness by contrasting cases where a function is composed of subfunctions with those where it is simply composed of triggers and effects.

The following section discusses subsidiary functions and this is followed by a discussion of how subsidiary functions are used to increase the expressiveness of the interpretation of simulation, resulting in a more precise and detailed design analysis report than is possible without their use. The theoretical basis for combination of subsidiary functions is then discussed.

The simplest case of functional decomposition is where a function depends on some expected combination of triggers and / or effects. For example, the recogniser

of a car headlamps main beam function might be described as follows.

```
lamp_switch_heads AND dip_switch_main
TRIGGERS
left_main_beam AND right_main_beam
```

Both trigger and effect have two elements both of which must be true for the function to be achieved. The use of logical `AND` means, of course, that if one of the headlamps fails to light in response to the switching on of the headlamps and switching to main beam, the trigger is true but the effect expression is false and so the function is said to have failed. This is consistent with Table 5.1 on page 92. In this case, then, the failure of either headlamp cannot be distinguished from the failure of both as either results in failure of the function. This is arguably appropriate in this case, as the loss of either headlamp renders the car unroadworthy, but in some cases the resulting design analysis report will be more precise and detailed if a distinction is drawn between failure of one of a function's effects and all of them. One approach to this is to link the effect with its own description of purpose. Where an effect can be associated with both its own purpose and with its own trigger it forms a subsidiary function, the subject of the next section. This allows greater refinement of the failure of a function compared to the simple binary Boolean true or false state of triggering and achievement of a function. Specifically, it allows the report to

- Distinguish between partial and total failure of a function in appropriate cases.
- Show the significance of alternative ways of achieving a function.
- Model differences between a function's purpose and its expected behaviour.

The first two of these advantages are discussed in this chapter, the remaining one is left until Section 7.5 in the following chapter.

One aspect of the use of logical operators that has not been raised is the rôle of `NOT`. One possible use is to distinguish effects that are inimical to achievement of a function as opposed to any effects that are irrelevant to a certain function. For example, the purpose of a headlamps dipped beam function is to light the road ahead without dazzling oncoming road users. Clearly, then, if a headlamp main beam stays on, this latter aspect of the purpose is not achieved. This is distinct from the case of the direction indicators, say, as they play no part in the purpose associated with the dipped beam function, so they could be on or off, while the main beams should be off. It is possible to specify this case using `NOT` like this.

```
lamp_switch_heads AND dip_switch_dipped
TRIGGERS
  left_dipped AND right_dipped
  AND NOT left_main_beam AND NOT right_main_beam
```

Therefore, of course, if either headlamp's main beam stays on, the function will have failed. In practice these cases need seldom be specified as any unwanted effects will be unexpected and as such will appear in the resulting report. The main beam staying on is not triggered so the effect will be achieved unexpectedly. As the consequences of unexpected achievement are associated with the effect itself, these consequences are noted even if only (in this case) one headlamp's main beam filament stays lit, so this will appear in the report independently of the dipped beam function. The other possible use of `NOT` is to simplify cases where a trigger or effect is not a binary state and one state is to be excluded. A possible example is a fan heater where the element will not come on unless the fan is running at either speed (slow or fast, or whatever range of speeds is supported), so the heating trigger might be specified as follows.

```
heater_switch_on AND NOT fan_stopped
```

This use of `NOT` also helps in mapping the functional description to a simulation that supports multiple levels of activity, as discussed in Chapter 11.

6.1 Subsidiary functions

While the idea of composing a function of triggers and effects combined using Boolean operators has been introduced, there are cases where a function is better decomposed in terms of subsidiary functions, each with its own trigger, effect and purpose, beside contributing to the achievement of the top-level function. One example of this might be the functional model of a hob where provided any ring works, the hob can be used for cooking. In other words a top level function `cook_on_hob` can be decomposed into four identical `cook_on_ring` functions, combined using `OR`. Each ring's behaviour is modelled in terms of function, because it has its own trigger (the knob being turned on), effect (output of heat for cooking) and purpose, the use of that ring for cooking. The benefit of this is that this allows the functional model to capture the idea that if any one ring works, this is a less serious failure than the loss of all the rings as some cooking is still possible by using the other rings, though the failure might restrict the sophistication of the cuisine.

This introduces the most important benefit of the two alternative decompositions of function, in terms of triggers and effects or in terms of subsidiary functions, which is that it increases the expressiveness of the language. It does this by enabling the description of cases where the failure of a subsidiary function has a different consequence from failure of the top level function. There are two cases where this typically applies. The first is where a system has alternative ways of achieving the purpose of the top level function, as in the hob example mentioned above. The other case is where the achievement of some of a function's expected effects can be regarded as mitigating the failure of a function and these can be distinguished from functions where the loss of any effect is regarded as tantamount to complete failure of the function. For example, a warning function might consist of both audible and visual signals (such as a horn and a telltale lamp). In this case, the failure of both means no warning is given while the failure of either one means there is some signal, so this can be regarded as a less serious failure of the system. This contrasts with functions where both effects can be regarded as tantamount to complete failure of the function. An example is be the headlamp function of a car already mentioned, where the loss of either the left or right headlamp renders the car unroadworthy. How the use of subsidiary functions increases the expressiveness of the language and hence of the automatically generated report is discussed in Section 6.2.

In addition, the use of subsidiary functions is consistent with the idea of a function as having a distinct purpose. In building a hierarchical functional description, the guiding principle is that if a distinct purpose can be identified with a subset of a function's effects, they can be modelled as a subsidiary function. For interpretation of design analysis, an important aspect of this is identifying effects whose failure has specific consequences, apart from those of the higher level function to which the effect contributes. This is because the resulting design analysis report will include entries for effects (or subsidiary functions) that are not present. If the report is to be as specific and detailed as possible, then what it should contain is the consequences associated with the actual missing function rather than a function higher in the hierarchy. To be consistent with the definition of function, a subsidiary function should be associated with a description of purpose. It should be noted, however, that even though a subsidiary function has its own purpose, its achievement contributes to fulfilment of the top level function's purpose, either as a component, as in the warning system, or an alternative, as in the hob example. It may well be the case that a system has several functions that, while related, are not part of one hierarchical functional description. This is discussed in Section 6.4.

Where a function is composed of subsidiary functions these take the place of the

trigger and effect expressions in the recogniser of the top level function. For example, a car's seat belt warning system that sounds a chimer and lights a dashboard telltale if either front seat is occupied and its seat belt is unbuckled might be described as follows.

```
FUNCTION belt_warning
  ACHIEVES warn_unbuckled
  BY
    FUNCTION audible_unbuckled_warning
    AND
    FUNCTION visual_unbuckled_warning
```

Here, the warning function consists of two subsidiary functions, each of which can be associated with a distinct purpose that contributes to correct fulfilment of the top level function's purpose. Each subsidiary function is a separate functional description. The audible warning function, for example, might be like this.

```
FUNCTION audible_unbuckled_warning
  ACHIEVES sound_unbuckled_warning
  BY
    driver_unbuckled OR passenger_unbuckled
    TRIGGERS
    sound_chimer
```

The labels for trigger and effect can then be used to attach system properties as described in the previous chapter. Note that the trigger expression has been simplified here. This system is used as a worked example in Chapter 11 and is discussed in more detail there. There is a description of the system itself in Section 11.1.1 on page 233. It will be appreciated that in this case both the subsidiary functions actually have identical triggers. An approach to eliminating this redundancy is introduced in the following chapter.

Each function is associated with its own description of purpose. This might be a component of the top level function's purpose, as here, but it could also be a more general purpose. For example, a plant monitoring system might use an audible warning to draw attention to the existence of any of several faults in the plant, while the nature of the fault is indicated by which of several warning lamps is lit. In this case the audible warning function might have a purpose description based on the need to draw attention to the presence of a fault and this function can be reused in the various warning functions, in combination with the correct warning lamp. The use of subsidiary functions and their associated descriptions of purpose is discussed in the following section.

6.2 Using subsidiary functions

This section discusses how subsidiary functions can be used to increase the expressiveness of the functional language and so of the automatically generated design analysis report that is the final result of the analysis. The discussion is mostly concerned with how this is used to describe failure of functions and subsidiary functions, that is the effect and consequences of the failure of a trigger to result in the expected effect. How functional decomposition relates to the unexpected achievement of the effects associated with a function is discussed in a separate subsection, Section 6.2.3.

That the hierarchical approach to function described in (Snooke & Price, 1998) can provide a more detailed explanation of the failure of a system function has been noted in the introduction to this chapter. The present approach builds on that work by allowing the decomposition of a function into either a combination of triggers and effects or into a combination of subsidiary functions, allowing those effects that achieve some purpose of their own to be distinguished from those that do not. The possibility of composing a function from subsidiary functions also supports the use of the language for support of functional refinement as part of the design process. In this section, the use of subsidiary functions to improve an automatically generated design analysis report will be discussed.

Two contrasting examples might be used to illustrate the use of subsidiary functions in design analysis. In a car's headlamp system, each headlamp being lit on main beam is, of course, necessary to achieve the "main beam" function. There is no benefit in complicating the description by attaching a function to each individual headlamp's output. There is never any reason to have one lit without the other and as the loss of either renders the car unroadworthy, there is arguably no benefit in distinguishing failures that result in the loss of one headlamp from those that result in the loss of both.

This can be contrasted with a system where the consequences of failures of one or other of the expected effects differ both from each other and from the failure of both, even though both are required for full achievement of the function. A possible example is a railway signalling system, where to direct an incoming train to a certain platform the signaller must first set the road so the train runs into the correct platform and then set the signals to allow the train driver to pull into the station. Suppose the points all worked correctly but the signals failed, the train could (subject to suitable operating checks being made) be called forward by hand (maybe using flags). This is clearly less difficult to handle than total failure of the system, so both points and signals fail. Associating the signal and point failures

with their own subsidiary functions allows the design analysis report to draw this distinction. Both of these examples are discussed in more detail below.

As a function can be composed of either effects or subsidiary functions combined using `AND` or `OR`, this gives four gradations of the effect a failure that one a function's effects has on the top level function. Each of these can be associated with a different style of entry in the resulting design analysis report.

6.2.1 Subsidiary functions with AND and OR

The simplest case is where a function depends on two effects that are not associated with any purpose of their own. This case is illustrated in Figure 6.1. Here,

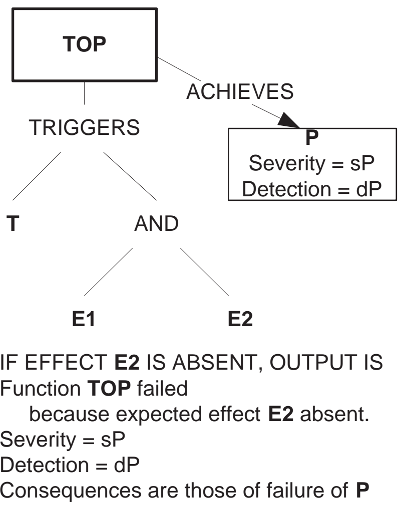


Figure 6.1: Combining effects using AND

failure of either effect will simply result in failure of the top level function and the consequences (together with values for severity and detection) are taken from that function, as they must be, there being no others available. Note that in this case, the expected effects share a trigger, though that itself might be an expression using Boolean operators. The headlamps example mentioned above is an example of this case. The main beam function could be specified as follows.

```
FUNCTION main_beam
  ACHIEVES light_road_ahead
  BY
    light_switch_heads AND dip_switch_main
    TRIGGERS
    left_headlamp_main AND right_headlamp_main
```

The associated description of purpose might look like this.

```
PURPOSE light_road_ahead
  DESCRIPTION "Lights road ahead so driver can see obstacles"
  FAILURE_CONSEQUENCE
      "Driver cannot see ahead. Legal implications"
    SEVERITY 8
    DETECTION 3
```

If some failure causes, say, the left headlamp to fail to light, the resulting design analysis report will include

> Function main_beam not achieved because expected effect left_headlamp_main was absent. Consequences are: Driver cannot see ahead. Legal implications. Severity = 8, detection = 3.

These are, of course, the consequences associated with the `light_road_ahead` purpose, associated with the `main_beam` function. As noted earlier, there is no need to specify any difference between the consequences of the loss of one headlamp and the loss of both, so the consequences and the severity and detection values are the same in both cases. There is therefore no need for the additional complexity in the functional description resulting from the use of subsidiary functions. It is appreciated that in practice the failure of one headlamp is perhaps less troublesome to the driver than that of both but the legal implications, if nothing else, make the example applicable.

As has already been discussed, this contrasts with the case where a function depends on two subsidiary functions. Here if one subsidiary function fails, the consequences in the resulting report (together with the values for severity and detection) are those of that subfunction. This is illustrated in Figure 6.2. In this case, the resulting report can either include consequences of failure of the top level function or the subsidiary functions. This can be illustrated using the railway signalling system mentioned in the introduction to this section.

```
FUNCTION accept_train
  ACHIEVES guide_to_platform
  BY
    FUNCTION set_points
    AND
    FUNCTION set_signals
```

The function might be associated with this description of purpose.

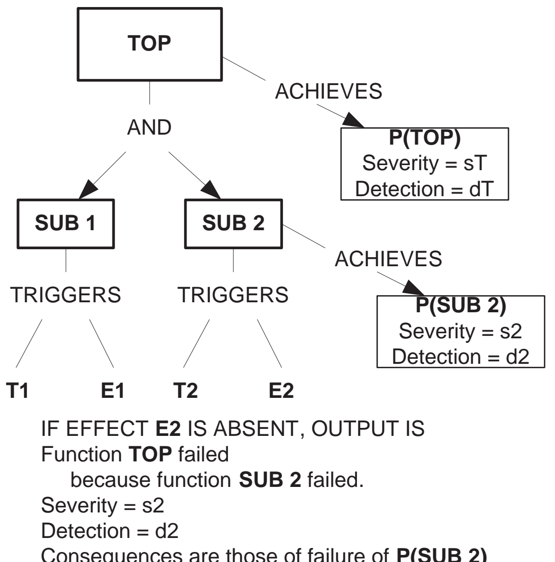


Figure 6.2: Combining subsidiary functions with AND

```
PURPOSE guide_to_platform
  DESCRIPTION "Allow a train to pull into the station platform"
  FAILURE_CONSEQUENCE "train cannot be accepted into station"
    SEVERITY 9
    DETECTION 3
```

Each subsidiary function has its own description, as a separate (reusable) data item. The `set_signals` function might have the following description.

```
FUNCTION set_signals
  ACHIEVES show_clear_road
  BY
    set_to_clear
    TRIGGERS
    signals_green
```

It has its own description of purpose.

```
PURPOSE show_clear_road
  DESCRIPTION "Allow driver to take train into platform"
  FAILURE_CONSEQUENCE "Need to flag train into platform"
    SEVERITY 7
    DETECTION 3
```

Now, if a power failure causes both the points and signals to fail, the resulting report might read

> Function accept_train has failed because functions set_points and set_signals have failed. Consequences are: Train cannot be accepted into station. Severity = 9, detection = 3.

The additional detail in the functional description allows the consequences of failure to only one of the subsidiary functions to be differentiated. If the signals fail but the points can be set, the report might read

> Function accept_train failed because function set_signals has failed. Consequences are: Need to flag train into platform. Severity = 7, detection 3.

So the consequences of the top level function are still included in the report if both the subsidiary figures fail (so the top level function can be thought of as having failed completely). However, if only one of the subsidiary functions fails, the consequences of failure of that function are included instead, though the report does, of course, still include an entry to the effect that the top level function has failed. There are arguable difficulties with this approach. The use of `AND` suggests that the top level function has failed and as such its consequences of failure should be included in the report. This is not done simply because the report should include the most specific results available. It will be seen that if the top level consequences are given, then there is no gain in information resulting from the use of subsidiary functions, a subsidiary function combined using `AND` will never have its consequences included in a report, as they will invariably be replaced by those of the top level function. In the approach adopted, the design analysis report does still include a note that the top level function has failed. This approach allows subsidiary functions to be used to describe cases where achievement of one of the subsidiary functions can be thought of as mitigating the failure of the top level function. There is more on such cases in Section 6.2.1.1.

Another use of subsidiary functions is where they constitute alternative ways of achieving a function, as in the hob example mentioned earlier. This case is illustrated in Figure 6.3. If one subsidiary function fails, there is no need for the report to mention the top level function as it need not have failed. Naturally in this case, then the consequences of failure of a subsidiary function are those associated with that function. So the hob might have the following functional description, simplified by assuming it only has two rings.

```
FUNCTION cook_on_hob
  ACHIEVES cook_food
```

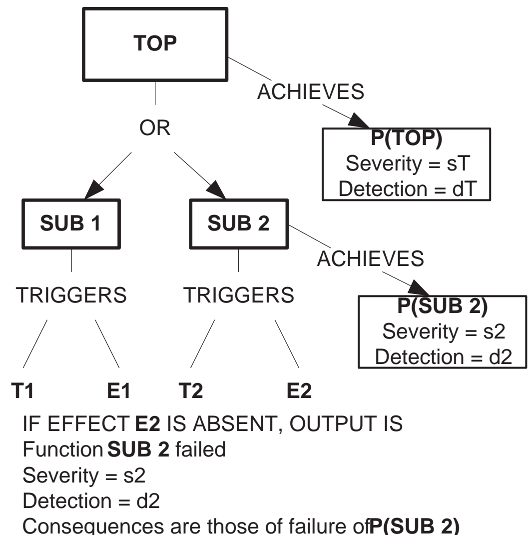


Figure 6.3: Combining subsidiary functions with OR

```
  BY
    FUNCTION cook_on_right
    OR
    FUNCTION cook_on_left
```

The function might be associated with this description of purpose.

```
PURPOSE cook_food
  DESCRIPTION "Allow preparation of cooked food"
  FAILURE_CONSEQUENCE "Hob cannot be used for cooking"
    SEVERITY 7
    DETECTION 3
```

Each ring has its own (identical) functional description.

```
FUNCTION cook_on_left
  ACHIEVES cook_on_ring
  BY
    switch_on
    TRIGGERS
    heat_ring
```

As the purpose of the two rings is identical and as descriptions of purpose are separate models, they can share this description of purpose.

```
PURPOSE cook_on_ring
  DESCRIPTION "Use ring for cooking"
  FAILURE_CONSEQUENCE "Ring cannot be used, limits cooking"
    SEVERITY 4
    DETECTION 3
```

If the left ring fails to come on when triggered, the report might read

> Function cook_on_left failed because expected effect heat_ring absent. Consequences are: Ring cannot be used, limits range of cooking. Severity = 4, detection = 3.

This is, of course, consistent with the use of OR in combining the subsidiary functions. The `cook_on_hob` function is achieved as the other ring works correctly. Naturally, if both rings' functions fail, the top level function also fails and the report will use that function's consequences.

One interesting feature of the functional model described here, and its description of purpose, is that it is not independent of the top level function. The consequence of its failure is merely the restriction on cooking. In other words this description of purpose can be written with the assumption that there is another ring so that failure of this one does not eliminate the possibility of using the hob. It will be seen that if the functional model was applied to a hob with only one ring, this would still not matter, as failure of the ring would lead to failure of the top level (`cook_on_hob`) function. There is, however, nothing to be gained by associating a subsidiary function with a function's one effect. This is consistent with the idea that a subsidiary function contributes to the purpose of the top level function rather than replacing it.

The final possible case is where a function depends on two effects combined using `OR`. Here the loss of one of the effects has no consequences at all, as there are no failed functions, as illustrated in Figure 6.4. This case is unlikely, simply because it will be unusual that a trigger will trigger either one or both of two possible effects, and the loss of one of the effect is of no consequence. A very weak example might be where a room has two lamps, both triggered by the same switch and either of which can be regarded as sufficient to light an occupant's way around the room. The functional description might look like this.

```
FUNCTION lights_on
  ACHIEVES light_room
  BY
    switch_on_lights
    TRIGGERS
    wall_lamp_on OR ceiling_lamp_on
```

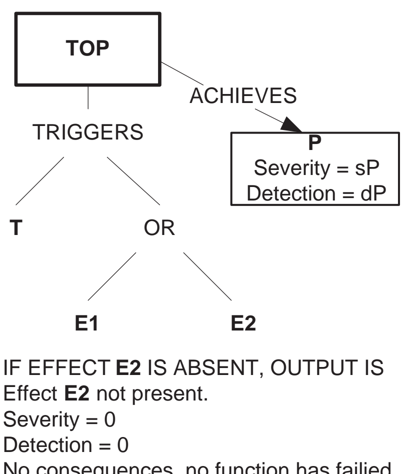


Figure 6.4: Combining effects with OR

The solitary description of purpose is as follows.

```
PURPOSE light_room
  DESCRIPTION "Allows use to see around the room"
  FAILURE_CONSEQUENCE "Room unlit, user may collide with furniture"
    SEVERITY 6
    DETECTION 2
```

Here, then, if one lamp fails, the report will include no more than

> Expected effect wall_lamp_on absent.

As this failure is not itself associated with a function, its failure is taken to be of no consequence. The contents of the report outlined here assume that both effects are expected which might not always be the case, it is not impossible that a given trigger might be intended to trigger one or both effects non-deterministically. It will be seen that the relation between trigger and effect is inappropriate as it suggests that the trigger triggers one or both of the possible effects, whereas both, of course, are expected. This highlights an interesting feature of this representation of function, which is that as it includes both the relationship between trigger and effect and effect and purpose, it arguably attempts to model two different (if related) aspects of the system, that is its purpose and the expected behaviour. Discussion of this is left until Section 7.5 on page 164.

The rule used for inclusion of consequences in the design analysis report is simple, in that where a function is composed of two (or more) subsidiary functions combined using either AND or OR, then where one of those subsidiary functions fails,

then the consequences of the failure of the subsidiary function are included in the report while if both (or all) of the subsidiary functions fail, then the consequences of the top level function's failure are included. The consequences are treated as a triplet of the description, and the values for severity and detection and these are not split, what is used in the resultant report comes from one description of purpose.

There is an apparent difficulty with this approach, which is that the effects of failure of a subsidiary function (or at least the consequences) are very similar whether `AND` or `OR` is used. The only difference is the note in the resulting report that the top level function has failed. As has already been discussed, it would be consistent with the use of `AND` to simply use the consequences (and value for severity) of the top level function given that that function is considered to have failed. This would mean that the consequences of subsidiary functions would never be used when they are combined using `AND` so the gain in expressiveness would not be realised. It is accepted that the functional description will be built with this in mind. For example, the consequences of failure of the signal subsidiary function in the example above implies that the points function worked correctly so the incoming train can reach the right platform.

Another aspect of the use of consequences of subsidiary functions is that the value for detection should be that of the subsidiary function, simply because this might be higher than that for the top level function, so the chosen approach avoids the need to use elements from different descriptions of purpose (not that doing so leads to any insuperable problems). For example, the failure of a function to make a car visible is (arguably) less severe if only the tail lamps fail, but harder to detect than if all the lamps fail.

It will be seen from the foregoing that subsidiary functions can be used to provide a finer distinction between degrees of achievement of a function than is possible with a straightforward Boolean description. This allows cases where a function is partly achieved to be distinguished from cases of total failure and also to indicate cases where alternative subsidiary functions allow a main function to be achieved.

6.2.1.1 Mitigation of failure

The use of the consequences of a subsidiary function in the design analysis report allows the functional modelling language to describe systems where partial achievement of a function is less severe than its complete failure, so achievement of one of a function's effects (subsidiary functions) can be thought of as mitigating the failure of the main function. The railway signalling example used earlier is

an example, in that it is more feasible to keep trains running if only the signals fail than if the signals and points fail, though it will clearly create considerable difficulties.

To reduce the apparent inconsistency with the use of `AND` in these cases, some thought was given to the idea of specifically labelling the subsidiary functions as mitigating the main function if achieved, so the top level signalling function might look like this.

```
FUNCTION accept_train
  ACHIEVES guide_to_platform
  BY
    MITIGATING FUNCTION set_points
    AND
    FUNCTION set_signals
```

All other parts of the functional description would be similar to the earlier example. The idea here would be that, because hand signals could be used if the signals failed, if the points work correctly, this mitigates the failure of the main function, so if the signals failed, a mitigating subsidiary function was achieved and the consequences of the failure of the signal subsidiary function would be included while if the points failed, no mitigating function was achieved so the consequences of the failure of the top level function would be used. The function would be labelled as `MITIGATING`, rather than using a "mitigated AND" operator to allow such asymmetric cases to be modelled. In the end this was not done as in practice it adds little expressiveness to the language. If the consequences of the failure of a subsidiary function really are identical to those of the top level function, then either they could be repeated in the subsidiary function's description of purpose or the two functions might share the same description of purpose. Another approach to this asymmetry in functional decomposition is to regard only the trigger and effect whose failure does not amount to total failure of the top level function (the signal function in this case) as a subsidiary function with its own purpose. This does lead to problems with the use of the top level function's `TRIGGERS` keyword as one half of the functional decomposition requires it and the other does not. How this can be solved is described in Section 7.2 in the next chapter.

6.2.2 Subsidiary functions and exclusive OR

Having looked at how the use of `AND` and `OR` can express a range of different relationships between a function and its expected effects, it seems worth discussing the place of exclusive OR in functional decomposition.

The rule used for inclusion of consequences of failure of a subsidiary function combined using AND or OR, use the subsidiary functions' consequences if one fails, the top level function's if both fail is not suitable with XOR as if both subsidiary functions are achieved, the top level function is inoperative. The relations between the state of a top level function and those of its subsidiary functions is described in Section 6.3 below. This relationship is more complex when XOR is used than either of the other operators. The rule for inclusion of consequences of failure is the reverse of that used with AND and OR, so if both subsidiary functions fail, the consequences of those failures, or possibly the worse of them, is included and the consequences of the top level function are included if one of the subsidiary functions fails.

In practice, it is suggested, XOR will rarely be used to combine subsidiary functions, though it is not impossible that a system has two mutually exclusive ways of achieving some purpose. One possible case where it might be is where there is a back up system that replaces the main system in event of a failure to that system. An emergency lighting system is a possible case in point. Here XOR might be used to specify that both systems' effects being present is not consistent with correct behaviour of the combined systems, so the emergency lights should not be lit when the main lights are working. Another possible use is to model software systems where there are two mutually exclusive ways of accessing data to preserve its consistency, such as database transactions. XOR is of more use for combining triggers (as in a lamp with two switches, such as a landing light).

The lack of any suitable example means that a discussion of the relationship between subsidiary functions and their consequences and the top level function is best left to the discussion of the theoretical basis for this relationships in Section 6.3.3.

6.2.3 Functional decomposition and unexpected effects

Having discussed the use of functional decomposition to improve the description of the effects of failure of a function it is time to consider the effects of the use of functional decomposition on unexpected achievement of the effects associated with a function.

As has already been described in Chapter 5, the consequences of unexpected achievement of an effect are associated with the effect itself (or, strictly, with the mapping between the functional description's effect and the appropriate system property) so that all unexpected effects are noted in the design analysis report, even where an effect does not amount to achieving a function. However, as noted

earlier in this chapter, it might be required to associate consequences of unexpected effects higher up a functional decomposition, typically because achieving all the effects of a function might result in different (more severe) consequences. For this reason, a function's description of purpose can, but need not, include a description of consequences of unexpected achievement of that function's required effects. This description of consequences will typically include values for severity and detection of the unexpected achievement of the effects. As noted earlier, this is why the tables in Section 6.3 have alternative consequences for unexpected achievement of a function.

The possibility of associating consequences of unexpected achievement of a function with that function entails the addition of rules for reporting of these consequences, following from rule 5.14 in Section 5.7. If f is a function, then if it includes unexpected consequences u_f then these will be reported (annotated $\mathrm{R}(u_f)$) if the unexpected effects are such that the effect expression of the function is true (that is the function is unexpected).

$$\mathrm{R}(u_f) \text{ if } \mathrm{Un}(f) \text{ and if } (f \text{ includes } u_f) \tag{6.1}$$

Where includes is used to show that the function is associated with its own unexpected consequences. This also entails additional conditions applying to the rule for reporting of an effects unexpected consequences, as these might be replaced by the function's consequences. The rule is that the unexpected effect's consequences are reported if the function is not unexpected (that is the effect expression as a whole is still false) or if the function has no unexpected consequences of its own. So unexpected consequences u_e of an effect e, are reported as follows.

$$\mathrm{R}(u_e) \text{ if } (\neg\mathrm{Tr}(f) \wedge e) \wedge (\neg\mathrm{Un}(f) \vee \neg(f \text{ includes } u_f)) \tag{6.2}$$

This is the complete version of rule 5.14 in Section 5.7.

An example of where this might be done is a car's stop lights, where the stop light function requires both stop lights to light in response to the trigger, pressing the brake pedal. It will be seen that in this case, the consequences of one brake light staying on (so being achieved unexpectedly) are less serious than those of both staying on, as if one light responds correctly to the trigger, then a following driver will still have some warning that the car is slowing while if both are lit the whole time, no warning is given.

The rules used for which consequences are included in the report are the same as that used for failure of a function. Where a function's effects (or subfunctions) are combined using `AND` or `OR`, if one effect is achieved unexpectedly, the unex-

pected consequences of that effect are included and if both effects of a function are achieved unexpectedly, then the consequences of unexpected achievement of the function are included, provided, of course, that the function's description of purpose has a set of unexpected consequences. The reverse applies if `XOR` is used, as is the case with failure of a function, so the function's unexpected consequences (if any) would be included if one but not both of the effects are unexpected.

This results in an apparent inconsistency in the case of unexpected achievement of a function where `OR` is used, as the function's effects might be achieved unexpectedly even if only one effect is achieved unexpectedly. This mirrors the similar effect with failure of a function using `AND`, where a function fails if one effect fails (provided, of course, that the effect is associated with a subsidiary function) but the consequences of that effect's subsidiary function are included in the report. The reason for accepting this anomaly is the same as that for the difficulty with `AND`, it leads to a more detailed report than if the consequences were dependent of achievement or otherwise of the function itself.

As the unexpected consequences are initially associated with an effect, there is, of course, no need for the effect to be associated with a subsidiary function. This reflects the idea that failure of a function is considered working down the hierarchy, while unexpected achievement works up the hierarchy.

Having considered the use of subsidiary functions in increasing the expressiveness of the functional language and so of the resulting report it is time to consider the relationships between the state of achievement of the top level function and the states of the subsidiary functions. This forms the topic of the next section.

6.3 A logical basis for subsidiary functions

Where a function is composed of subsidiary functions, these are combined using the logical operators in much the same way as triggers and effects are. Where subsidiary functions are used, each will have its own trigger and effect, so a subsidiary function can be in any of the four functional states illustrated in Table 5.1 on page 92. Therefore the state of the top level function must be defined in terms of the states of the subsidiary functions. The rule used is that the truth of the trigger of the top level function depends on the truth of the triggers of the subsidiary functions combined using the operator that relates the subsidiary function and the truth of the effect of the top level function is derived from the truth of the effects of the subsidiary functions related using that operator.

To put this more formally, let f be a function composed of two subsidiary functions a and b related using some Boolean operator $\otimes$ so f is composed of $a \otimes b$. The rules for finding the state of f are as follows.

$$\text{Tr}(f) \text{ if } \text{Tr}(a) \otimes \text{Tr}(b) \tag{6.3}$$

$$\text{Ef}(f) \text{ if } \text{Ef}(a) \otimes \text{Ef}(b) \tag{6.4}$$

The function state can then be derived from these using rules listed in Appendix A as A.7 to A.10.

It will be seen that this forces the same Boolean relation between the triggers and effects of the subsidiary function. This limitation can be circumvented by using trigger and effect expressions instead of the subsidiary functions (with the loss of associated descriptions of purpose for the lower level functions). If the descriptions of purpose are required, then incomplete functions, described in the following chapter, can be used.

For example, if a function F depends on subsidiary functions *C1 AND C2* the top level function is only triggered if the triggers for both subsidiary functions are true and only achieved if both the subsidiary functions' effects are present. If only one subsidiary function is triggered, the implicit trigger expression for F is not true. Likewise if that subsidiary function's effect is present, the effect of F is still not true as it requires both. Therefore F is `inoperative` in this case. The results of this approach are summarised in table 6.1. where the subsidiary

subsidiary functions		main function		
C 1	C 2	C 1 AND C 2	C 1 OR C 2	C 1 XOR C 2
Inoperative	Inoperative	**Inoperative**	**Inoperative**	**Inoperative**
Inoperative	Achieved	**(Inoperative)**	**Achieved**	**Achieved**
Inoperative	Failed	**(Inoperative)**	**Failed**	**Failed**
Inoperative	Unexpected	**(Inoperative)**	**Unexpected**	**Unexpected**
Achieved	Achieved	**Achieved**	**Achieved**	**(Inoperative)**
Achieved	Failed	**Failed**	**(Achieved)**	**Unexpected**
Achieved	Unexpected	**Unexpected**	**(Achieved)**	**Failed**
Failed	Failed	**Failed**	**Failed**	**(Inoperative)**
Failed	Unexpected	**(Inoperative)**	**(Achieved)**	**(Achieved)**
Unexpected	Unexpected	**Unexpected**	**Unexpected**	**(Inoperative)**

Table 6.1: Achievement of a function in terms of subsidiary functions.

functions are **Child 1** and **Child 2** and are related using the binary Boolean

operators. It will be seen that some of the results are, perhaps unexpected, or at least not intuitively correct. For this reason, fuller descriptions of the relationships are given in following subsections, with brief discussions of the more surprising results. However, the approach taken is consistent, both with itself and with the idea that the four functional states are defined in terms of the truth of triggers and effects.

In some cases (identified in the table by the resulting functional state being in parentheses) a top level function might be in a state consistent with correct operation of the system despite the fact that a subsidiary function is either failed or achieved unexpectedly. In these cases the resulting design analysis report need not include any reference to the top level function, so where, for example, a function depends on two subsidiary functions combined using `AND`, if one subsidiary function has failed and the other is inoperative, the report need only include an entry for the failed subsidiary function. These points will, it is hoped be clarified by the following subsections, which discuss the relationships between functions and subsidiary functions for each logical operator.

The reporting of subsidiary functions follows the rule for reporting of functions, rule A.11 in Appendix A. However, as described earlier, to capture the idea that partial achievement of a function might mitigate its failure there are different rules for reporting the consequences of failure of subsidiary functions. If f is a function with its consequences of failure c_f and composed of subsidiary functions a and b, combined using `AND` or `OR`, and each with their own consequences of failure c_a and c_b then the consequences of failure of f are reported ($\mathrm{R}(c_f)$) only if both subsidiary functions fail.

$$\mathrm{R}(c_f) \text{ if and only if } \mathrm{Fa}(a) \wedge \mathrm{Fa}(b) \tag{6.5}$$

If only one of the subsidiary functions, say a fails then its consequences are reported, even if this means that f has failed.

$$\mathrm{R}(c_a) \text{ if } \mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b) \tag{6.6}$$

Where the subsidiary functions are combined with `XOR`, this rule is inappropriate and instead we report the consequences of failure of f if only one of the subsidiary functions fails.

$$\mathrm{R}(c_f) \text{ if } (\mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b)) \vee (\neg\mathrm{Fa}(a) \wedge \mathrm{Fa}(b)) \tag{6.7}$$

The rules for reporting consequences of unexpected achievement of functions follow the rules numbered A.14 and A.15 in Appendix A, as described earlier in the thesis.

These reporting rules are illustrated in the tables that accompany the following sections.

6.3.1 Subsidiary functions and AND

The relationships between the states of a function and its subsidiary functions, combined using logical `AND` are shown in Table 6.2. This table (as do the similar

function *C1*			function *C2*			*F* (*C1* AND *C2*)			generated text	consequences
t1	*e1*	state	*t2*	*e2*	state	t	e	state		
F	F	In	F	F	In	**F**	**F**	**In**	(no entry)	
F	F	In	T	T	Ac	**F**	**F**	**In**	Function *C2* achieved	
F	F	In	T	F	Fa	**F**	**F**	**In**	Function *C2* failed because expected effect *e2* absent	Fa(*C2*)
F	F	In	F	T	Un	**F**	**F**	**In**	Function *C2* achieved unexpectedly because unexpected effect *e2* present	Un(*C2*) or Un(*e2*)
T	T	Ac	T	T	Ac	**T**	**T**	**Ac**	Function *F* achieved	
T	T	Ac	T	F	Fa	**T**	**F**	**Fa**	Function *F* failed because function *C2* failed	Fa(*C2*)
T	T	Ac	F	T	Un	**F**	**T**	**Un**	Function *F* achieved unexpectedly because function *C2* achieved unexpectedly	Un(*C2*) or Un(*e2*)
T	F	Fa	T	F	Fa	**T**	**F**	**Fa**	Function *F* failed because functions *C1* and *C2* failed	Fa(*F*)
T	F	Fa	F	T	Un	**F**	**F**	**In**	Function *C1* failed and function *C2* achieved unexpectedly	max (Fa(*C1*), Un(*C2*))
F	T	Un	F	T	Un	**F**	**T**	**Un**	Function *F* achieved unexpectedly because functions *C1* and *C2* achieved unexpectedly	Un(*F*) or max (Un(*C1*), Un(*C2*))

Table 6.2: Functional decomposition using AND

tables in the following subsections) makes explicit the relationship between the functional state (both of the top level function and the subsidiary functions) and the trigger and effect. It also includes a brief rendition of the text that will be included in a design analysis report and a brief notation showing how the report will include the consequences of the functional state. There is a fuller description of the notation used in these tables in Appendix D.

It will be seen that the top level function cannot be in a state that requires its trigger to be true unless both subsidiary functions' functions are true and also that it cannot be in a state in which its effects are true unless both subsidiary

functions' effects are true. This is what would be expected, the top level function is only achieved if both subsidiary functions are achieved.

It will be seen that where the top level function is in a state apparently consistent with correct behaviour of the system (in this case inoperative) but one of the subsidiary functions is not (it has failed or is unexpected) then the generated text (for the design analysis report) need only refer to that subsidiary function. In such cases, both in this table and in Table 6.1 above, the top level function's state is in parentheses to highlight this. Where the top level function has failed because of the failure of one subsidiary function (in this case *C2*) the report can include this detail and also include a specific set of consequences. The report uses the subsidiary function's consequences in cases where one of the subsidiary functions is inconsistent (failed or unexpected) and uses the consequences of the top level function where both subsidiary functions have failed or are unexpected. This allows cases where achievement of one of the subsidiary functions mitigates failure of the top level function to be distinguished, which is one of the advantages of using subsidiary functions. As has been discussed, this representation of consequences is (arguably) inconsistent with `AND` for failure of the top level function but if it is not done, the expressiveness of the approach is reduced. The generated text does include a note that the top level functional has failed. The table includes alternatives for the consequences of unexpected achievement of functions, simply because such consequences are required for individual effects but might not be used higher up the decomposition. In such cases, the report will include the highest available set of unexpected consequences.

It should be noted that the use of subsidiary functions is different from a function that requires a combination of triggers and effects, as each subsidiary function's trigger is associated with that function's effect. For example, if the hob's functional model did not use subsidiary functions, and the rings and their switches are simply treated as triggers and effects of the `cook_on_hob` function, there is no way of associating the switch with the correct ring. If either of the switches is on and either of the rings is on, the function is achieved. This is another reason for adopting subsidiary functions and an approach to using subsidiary functions to associate triggers and effects where they are not to be associated with their own descriptions of purpose is described in Section 7.2.

6.3.2 Subsidiary functions and OR

When the subsidiary functions are combined using `OR` then, following the rule that the trigger and effect of the top level function depends on the triggers and effects,

respectively, of the subsidiary functions, if either trigger is true, the top level function is triggered, as shown in Table 6.3. Here, of course a top level function

function *C1*			function *C2*			*F* (*C1* OR *C2*)			generated text	consequences
t1	*e1*	state	*t2*	*e2*	state	t	e	state		
F	F	In	F	F	In	**F**	**F**	**In**	(no entry)	
F	F	In	T	T	Ac	**T**	**T**	**Ac**	Function *F* achieved because function *C2* achieved	
F	F	In	T	F	Fa	**T**	**F**	**Fa**	Function *F* failed because function *C2* failed	Fa(*C2*)
F	F	In	F	T	Un	**F**	**T**	**Un**	Function *F* achieved unexpectedly because function *C2* achieved unexpectedly	Un(*C2*) or Un(*e2*)
T	T	Ac	T	T	Ac	**T**	**T**	**Ac**	Function *F* achieved	
T	T	Ac	T	F	Fa	**T**	**T**	**Ac**	Function *C2* failed because expected effect *e2* absent	Fa(*C2*)
T	T	Ac	F	T	Un	**T**	**T**	**Ac**	Function *C2* achieved unexpectedly because unexpected effect *e2* present	Un(*C2*) or Un(*e2*)
T	F	Fa	T	F	Fa	**T**	**F**	**Fa**	Function *F* failed because functions *C1* and *C2* failed	Fa(*F*)
T	F	Fa	F	T	Un	**T**	**T**	**Ac**	Function *C1* failed and function *C2* achieved unexpectedly	max (Fa(*C1*), Un(*C2*))
F	T	Un	F	T	Un	**F**	**T**	**Un**	Function *F* achieved unexpectedly because functions *C1* and *C2* achieved unexpectedly	U(F) or max (Un(*C1*), Un(*C2*))

Table 6.3: Functional decomposition using OR

might be achieved despite failure of a subsidiary function, as in the sixth line of the table. In such cases, the report need only refer to the failure of the subsidiary function. The result of one subsidiary function failing and the other being achieved unexpectedly (the ninth row of the table) is apparently anomalous. However the result is consistent with the use of `OR`, in that the top level function is triggered (one of the subsidiary functions is) and its effect is true (one of the subsidiary function's effect is true). In practice, too, this is a plausible result, in that the purpose of the top level function can be fulfilled in this case. To demonstrate, consider the hob example mentioned earlier. Suppose that a connection error in an electric hob means that turning on the left front ring results in the right front ring heating. Clearly in this case, the left front ring's function is failed (triggered but not effective) and the right front ring's function is unexpected (not triggered but effective). In this case, then a top level function for the hob as a whole is achieved and, at least arguably, this is the on the grounds that this state allows cooking to be done so its purpose is fulfilled. This is subject to the proviso that the

cook notices that the wrong ring has come on. The precise effect of this will depend on the type of hob, of course, as this state is fairly obvious with radiant rings, less so with hot plates. This does leave the model builder to build the functional model with this in mind, of course. This apparent anomaly is mitigated by the fact that the generated text will refer to the inconsistent behaviour of the subsidiary functions, as shown in the table.

The same rule for deciding which consequences are included in the report is used here, so where one subsidiary function has failed (or is unexpected), its consequences are used, while if both have failed (or are unexpected) the top level function's consequences are used. There is a similar inconsistency as is the case with `AND` here, in that where one of the subsidiary functions is unexpected, so is the top level function but the subsidiary function's consequences are included in the report. This is for the same reason, so that the report distinguishes between cases where one or both subsidiary functions are unexpected.

6.3.3 Subsidiary functions and XOR

For a top level function using exclusive or (`XOR`) to be achieved, one but not both of the subsidiary functions must be achieved, as shown in Table 6.4. In fact, of course, the same unexpected result of one subfunction being failed and the other unexpected as with `OR` is apparent here, and the same argument apply. It will be seen that there are other apparently unexpected results here, such as one subfunction's inconsistent behaviour causing the other inconsistent behaviour in the top level function, so that in the seventh row of the table, an unexpected subsidiary function causes the top level function to fail. This is consistent with the approach taken, as the trigger expressions combined using the operator resolve to false, while the effect resolves to true. The opposite case is found in the sixth row. It is suggested that while these results appear anomalous, they are therefore correct and the report will be consistent with this.

As noted earlier, the rule used with `AND` and `OR` for including consequences in the report (use the subsidiary function's consequences if subsidiary function has failed or is unexpected, the top level functions if both are) is clearly inappropriate here, as it is only when one subsidiary function has failed (or is unexpected) that the top level function fails (or is inconsistent). Therefore, for `XOR`, the rule is reversed and the consequences of the top level function are included in the report when both subsidiary functions have failed or are unexpected and the consequences of the top level function when only one has. The approach taken here is that the report will include the consequences of that subsidiary function that has the more

function *C1*			function *C2*			*F* (*C1* XOR *C2*)			generated text	consequences
t1	*e1*	state	*t2*	*e2*	state	t	e	state		
F	F	In	F	F	In	**F**	**F**	**In**	(no entry)	
F	F	In	T	T	Ac	**T**	**T**	**Ac**	Function *F* achieved because function *C2* achieved	
F	F	In	T	F	Fa	**T**	**F**	**Fa**	Function *F* failed because function *C2* failed	Fa(*F*)
F	F	In	F	T	Un	**F**	**T**	**Un**	Function *F* achieved unexpectedly because function *C2* achieved unexpectedly	Un(*F*) or Un(*C2*)
T	T	Ac	T	T	Ac	**F**	**F**	**In**	Functions *C1* and *C2* achieved	
T	T	Ac	T	F	Fa	**F**	**T**	**Un**	Function *F* achieved unexpectedly because function *C2* failed	Un(*F*) or Fa(*C1*)
T	T	Ac	F	T	Un	**T**	**F**	**Fa**	Function *F* failed because function *C2* achieved unexpectedly	Fa(*F*)
T	F	Fa	T	F	Fa	**F**	**F**	**In**	Functions *C1* and *C2* failed	max (Fa(*C1*), Fa(*C2*))
T	F	Fa	F	T	Un	**T**	**T**	**Ac**	Function *C1* failed and function *C2* achieved unexpectedly	max (Fa(*C1*), Un(*C2*))
F	T	Un	F	T	Un	**F**	**F**	**In**	Functions *C1* and *C2* achieved unexpectedly	max (Un(*C1*), Un(*C2*))

Table 6.4: Functional decomposition using XOR

severe consequences (indicated by 'max' in the table) but, of course, both could be included.

6.4 Relationships between system and function

It seems worth closing this chapter with a brief discussion of the relationship between the functional hierarchy and the system. Figure 6.5 shows the relationships between the functional descriptions, descriptions of purpose and system attributes for the hob example used earlier. In this case, a given ring's function can be associated with a subsystem of the hob in that not all the components of the hob are involved in achieving that function. For example the failure of a wire connecting the left ring with its knob does not affect achievement of the cook on right ring function. However, there is unlikely to be a neat mapping between a functional subsystem and a structural subsystem. Typically, the failure of some components will affect several of a system's functions, of course and it is possible that a physical subsystem will contribute to several functional subsystems, such as the handset of

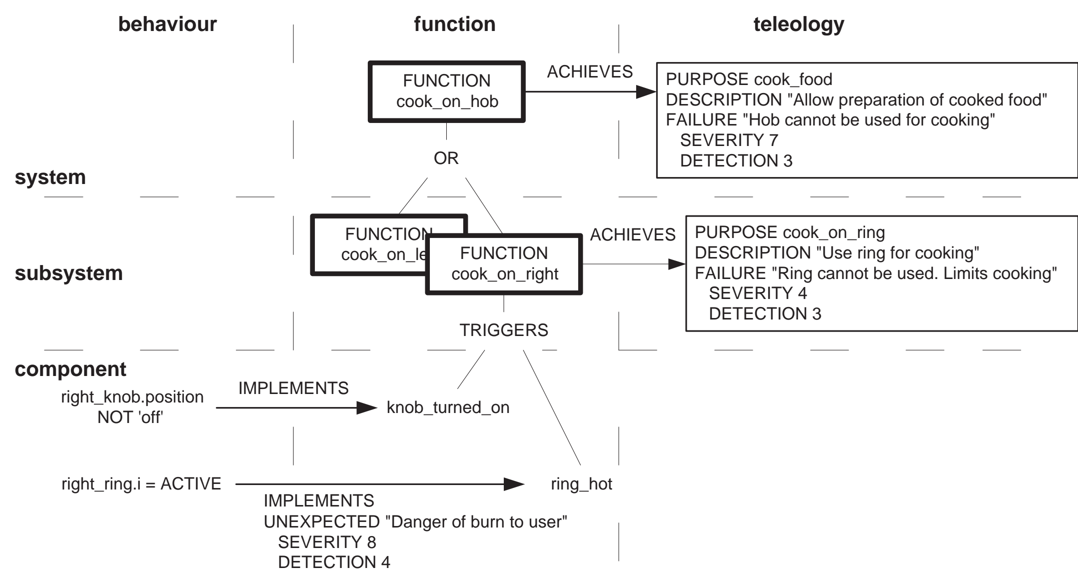


Figure 6.5: Model relationships for the hob example.

a telephone being required in both the transmit and receive functions.

A further area to consider when discussing functional decomposition is the point that the top level of a functional hierarchy relates to a purpose not to a physical system. Some systems might well be cohesive enough to be analysed as a whole even though they actually fulfil several different (if related) functions. For example, it is quite likely that a car's exterior lighting system will be analysed as a whole as it is a cohesive structural design, but this system implements several distinct functions, each fulfilling a distinct purpose. Some of these are listed in Table 6.5. The functions informally listed in that table do not combine to achieve

function	effect
make car visible	left and right side lights and tail lights lit
light road	left and right headlamps main beam
light road without dazzling	left and right headlamps dipped
warn of slowing	left and right brake lights lit
indicate right	right hand indicators flashing
indicate left	left hand indicators flashing

Table 6.5: Functions associated with a car's exterior lights.

some higher level purpose, with the proviso that dipped headlamps might be considered to contribute to the "make car visible" function. Therefore, they should not be modelled as components of a system level function. That there is no need

for a system level "exterior lights" function is made clear by the fact that there is no single purpose that can be attached to it, other than the rather loose idea that the car is not roadworthy if any of the lights is not working and this is captured by the need to fulfil all the listed purposes.

This serves to highlight the idea that a functional hierarchy should have a clearly defined purpose at the top and need not imply the correct working of the whole system, as there is no necessary relationship between the structural and functional decompositions of the system. Of course, if the system has parts that are not necessary to achieve any function, these are likely to be redundant, though they could of course be associated with fault tolerant back up functionality.

Another point that arises is that the listing of functions in Table 6.5 is only one possible alternative functional description. An alternative might have two "make visible" functions, a parked one requiring sidelights and tail lights and a running one that adds dipped or main beam headlamps. This does suggest that the building of an appropriate functional model requires some knowledgeable judgement as to the correct relationships between the different functions and their triggers and effects. However, this could be seen as emphasising the advantage of this approach in that using functional description to interpret the results of a simulation for design analysis does mean that these relationships need only be described once, rather than repeating the description each time a given failure is found, as would be the case if only the simulation were available.

The alternative decomposition, with the dipped beam contributing to both the make visible and the light road ahead functions introduces the idea that a subsidiary function might be associated with more than one top level functions. As each function is a separate model, there is no difficulty in doing so. A possible example is the plant monitoring audible warning function used as an example earlier. As this same siren is used to draw attention to any one of a number of plant failures, the effect might well be treated as a subsidiary function of each failure's warning function. It will be appreciated that there is a possibly drawback in doing so, in that the severity of the failure of the warning siren might well be considered to depend on which plant failure it is failing to draw attention to (so to which top level function its failure is affecting). It might be better to use several subsidiary functions and associate each audible warning subsidiary function with the same effect (the siren) or even to dispense with subsidiary functions altogether in such cases.

This chapter has described the decomposition of system functions in cases where the subsidiary functions are themselves complete representations of function, with trigger, effect and reference to a description of purpose. However, there are cases

where a system function is better decomposed in to several subsidiary functions that share one of these three elements. The next chapter considers such cases, and the use of such incomplete representations of function.

Chapter 7

Incomplete representations of function

While a subsidiary function might have its complete set of three characteristic elements (the trigger, effect and purpose), as described in the previous chapter, there are cases where one of these elements is common to some function's subsidiary functions and so the repetition of this element is redundant. The seat belt warning example in Section 6.1 is a case in point as the audible and visual warning subsidiary functions will share a trigger. This suggests a rôle for incompletely specified functions, allowing the repetition of common elements to be avoided. Indeed, such representations are necessary for completeness of the language. The use of Boolean expressions to relate subsidiary functions (as discussed in the previous chapter) limits such decompositions to cases where the same operator is used for the triggers and effects of the subsidiary functions. The use of incomplete representations of function allows functions whose triggers and effects are not to be combined using the same operator to be related as subsidiary functions, with their own descriptions of purpose.

The use of three elements in a complete function introduces the possibility of incomplete functions (or incomplete representations of function) that specify the relationship between any two of these elements. If function is seen as a relational concept, then at least two elements are necessary, of course, so the function can define the relation between them. The three possible pairings of elements are:-

- The relation between effect and purpose.
- The relation between trigger and effect.
- The relation between trigger and purpose.

This in turn suggests that there are three possible classes of incomplete function.

Purposive incomplete function (abbreviated to PIF) relates an effect to a purpose. It is used where several effects each have their own purpose (besides contributing to that of a higher level function), but share the trigger of that higher level function.

Operational incomplete function (OIF) relates a trigger to an effect with no separate purpose. It is used to associate specific triggers with specific effects, all of which contribute a higher level function

Triggered incomplete function (TIF) relates a trigger to a purpose. It can be used where some common effect can be triggered in different ways and where the trigger defines the purpose of the effect and so the function.

Each of these classes of incomplete function will be discussed in its own section, before concluding the chapter with a discussion of how these incomplete functions are used in differing functional decompositions.

The terminology used for the first two classes of incomplete function follows the alternative definitions of function in (Chittaro & Kumar, 1998). A purposive incomplete function expresses the relationship between behaviour (specifically effect) and purpose, consistently with that paper's purposive definition of function while an operational incomplete function describes the relationship between trigger and effect (so input and output), consistently with their operational definition of function. The triggered incomplete function does not follow so definitely from a definition of function and there might be felt to be no need for that class of incomplete function as there is no direct relationship between trigger and purpose — a trigger must result in an effect to achieve its purpose. The relationships between trigger, effect and purpose are linear in that a trigger results in an effect that fulfils a purpose. Earlier discussions of this approach to functional decomposition, (Bell *et al.*, 2005a; Bell *et al.*, 2005b) did indeed only include discussion of purposive and operational incomplete functions. However, there are reasons for the inclusion of the remaining class, to be discussed in Section 7.3

7.1 Purposive incomplete function

A purposive incomplete function (PIF) maps between an effect expression (which might, of course, combine individual effects using Boolean operators) and a specific purpose. There is no trigger expression, allowing purposive incomplete functions

to be used to describe subsidiary functions that share a triggering condition. This avoids the redundant repetition of triggers noted in the subsidiary function example mentioned earlier, the seat belt warning system. That same functional description can be built using purposive incomplete functions instead of the (complete) functions used in the earlier example.

```
FUNCTION belt_warning
  ACHIEVES warn_unbuckled
  BY
    driver_unbuckled OR passenger_unbuckled
    TRIGGERS
      PIF audible_unbuckled_warning
      AND
      PIF visual_unbuckled_warning
```

Each purposive incomplete subfunction is a separate model wherein the recogniser is reduced to an expression describing the effect. The reference to a description of purpose is similar to that for a complete function. It will therefore have a set of consequences of failure associated with it (and so with the purposive incomplete function) and might also have a set of consequences of unexpected achievement of the effect, in exactly the same way as a description of purpose associated with a complete function. The `audible_unbuckled_warning` subfunction as a purposive incomplete function might look like this.

```
PIF audible_unbuckled_warning
  ACHIEVES sound_unbuckled_warning
  BY
    sound_chimer
```

The difference between a purposive incomplete function and a straightforward effect is the association with some specific purpose that contributes to that of the top level function of which the PIF is a subsidiary function. A purposive incomplete function can be thought of as inheriting a trigger from the function of which it is a subsidiary function. This avoids the need to repeat an identical expression for several subfunctions, avoiding the danger of inconsistency between the expressions and also simplifying the mapping between system properties and trigger, as only one such mapping is needed.

The relationship between trigger, effect and achievement of a purposive incomplete function is similar to that for a complete function, the only difference being that the trigger is elsewhere in the function hierarchy. If the effect is true and the

trigger false, the effect is unexpected. Whether the effect of the function of which the PIF is a subfunction is also achieved unexpectedly will depend on the relationship between the function's subsidiary functions and which of the subsidiary PIFs' effects are present. To define the relationship between a function and its subsidiary PIFs more formally, let f be a system function with subsidiary PIFs p and q, related using some binary Boolean operator $\otimes$.

$$\mathrm{In}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \neg(\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{7.1}$$

$$\mathrm{Fa}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \neg(\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{7.2}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge (\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{7.3}$$

$$\mathrm{Ac}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge (\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{7.4}$$

The trigger expression belongs to f, so we can consider f to be triggered. Notice that the trigger cannot be associated with each of the PIFs (making them complete subsidiary functions) as if they are combined using `XOR` then teh top level function will never be triggered as both subsidiary functions' trigger expressions will be either true of false at any one time. This relationship is discussed further below.

In the seat belt warning system example, if the audible warning sounds despite there being no unbuckled passengers then the seat belt warning function's effects are not achieved unexpectedly because of the lack of the visual signal. Similarly, if the purposive incomplete function's inherited trigger is true and its effect false, the PIF has failed and whether the top level function has failed will, likewise, depend on the state of other subsidiary functions and the relationship between them. In this case, failure of either the audible or visual warning PIFs will amount to failure of the belt warning function. However, it could be argued that the achievement of either of the subsidiary functions mitigates failure of the warning function as at least some indication is given, so the failure of the warning system might be felt to be less severe than if both were lost. This is consistent with the idea of using subsidiary functions to show cases where achievement of some of a top level function's effects mitigates failure of the function, as discussed in the previous chapter.

The relationships between the state of a top level function and the states of its purposive incomplete subsidiary functions are worth a brief description. The simplest and most common cases is where the subsidiary PIFs are combined using `AND`, as in the seat belt warning system discussed above, and the resulting relationships are shown in Table 7.1. When purposive incomplete functions are used, the trigger is a Boolean expression belonging to the top level function, so is not

function *F* (*P1* AND *P2*)							generated text	consequences
t	*P1*		*P2*					
	e1	state	*e2*	state	e*F*	state		
F	F	In	F	In	**F**	**In**	(no entry)	
F	T	Un	F	In	**F**	**In**	Function *P1* achieved unexpectedly because unexpected effect *e1* present	Un(*P1*) or Un(*e1*)
F	T	Un	T	Un	**T**	**Un**	Function *F* achieved unexpectedly because functions *P1* and *P2* achieved unexpectedly	Un(*F*) or max (Un(*P1*), Un(*P2*))
T	F	Fa	F	Fa	**F**	**Fa**	Function *F* failed because functions *P1* and *P2* failed	Fa(*F*)
T	T	Ac	F	Fa	**F**	**Fa**	Function *F* failed because function *P2* failed	Fa(*P2*)
T	T	Ac	T	Ac	**T**	**Ac**	Function *F* achieved	

Table 7.1: Decomposition using partial incomplete functions with AND

derived from the truth of the subsidiary functions triggers as is the case where they are complete functions. Where `AND` is used to relate the subsidiary functions, the results are similar to those obtained using complete subsidiary functions, shown in Table 6.2 on page 138, but without the need to include relationships where only one of the subsidiary functions is triggered, of course.

The same relationship between the use of purposive incomplete functions and complete subsidiary functions is also found where `OR` is used to combine the subsidiary functions, as shown in Table 7.2, which can be compared with Table 6.3 on page 140. One possible example of the use of PIFs with `OR` is a room lighting

function *F* (*P1* OR *P2*)							generated text	consequences
t	*P1*		*P2*					
	e1	state	*e2*	state	e*F*	state		
F	F	In	F	In	**F**	**In**	(no entry)	
F	T	Un	F	In	**T**	**Un**	Function *F* achieved unexpectedly because function *P1* achieved unexpectedly	Un(*P1*)
F	T	Un	T	Un	**T**	**Un**	Function *F* achieved unexpectedly because functions *P1* and *P2* achieved unexpectedly	Un(*F*) or max (Un(*P1*), Un(*P2*))
T	F	Fa	F	Fa	**F**	**Fa**	Function *F* failed because functions *P1* and *P2* failed	Fa(*F*)
T	T	Ac	F	Fa	**T**	**Ac**	Function *P2* failed	Fa(*P2*)
T	T	Ac	T	Ac	**T**	**Ac**	Function *F* achieved	

Table 7.2: Decomposition using partial incomplete functions with OR

circuit where a switch is intended to switch on two lamps, either of which is suf-

ficient to light an occupant's way around the room. For example, if there was a ceiling lamp and a wall lamp, the functional description might look like this.

```
FUNCTION lights_on
  ACHIEVES light_room
  BY
    light_switch_on
    TRIGGERS
      PIF ceiling_lamp
      OR
      PIF wall_lamp
```

The wall lamp's function might be as follows.

```
PIF wall_lamp
  ACHIEVES light_side_of_room
  BY
    wall_lamp_on
```

Because each light is associated with its own function, the failure of one light will be reported, using that PIF's description of purpose and consequences of failure. For example, if the wall lamp failed the report might read

> Function wall_lamp failed because expected wall_lamp_on not present. Consequences are side of room dimly lit. Severity 3, detection 2.

The consequences are, of course, those of the description of purpose associated with the wall lamp function. This would clearly contrast in severity with failure of both lamps leaving the occupant in the dark. If the lamps were simply effects associated with the `lights_on` function, then if one failed the report would show nothing, unless, of course, `AND` was used instead of `OR`, in which case the report would show the room was not lit and could not distinguish between failure of one lamp or failure of both.

This raises the interesting point that the relations between subsidiary functions' purposes might differ from the system's expected behaviour. In addition to enabling the report to show a more refined interpretation of the system behaviour than a simple Boolean ("all or nothing") functional description would allow, the use if purposive incomplete functions allows the functional description to capture differences between the intended purpose of a system and its expected behaviour. In this case, the (main) purpose of the lamps differs from the expected behaviour

of the system as the purpose is adequately fulfilled by one of the lamps, while both are, of course expected to light up in response to the trigger. The idea that a function represents both a relationship between behaviour and purpose and the expected behaviour of a system is discussed further in Section 7.5.

The similarity between the results of using complete functions (repeating the trigger) and purposive incomplete functions is lost when the relation is `exclusive OR`. Table 7.3 shows that achievement of the top level function depends on the trigger being true and one (but, of course, not both) of the PIFs being achieved. If

function *F* (*P1* XOR *P2*)							generated text	consequences
	P1		*P2*					
t	*e1*	state	*e2*	state	e*F*	state		
F	F	In	F	In	**F**	**In**	(no entry)	
F	T	Un	F	In	**T**	**Un**	Function *F* achieved unexpectedly because function *P1* achieved unexpectedly	Un(*F*) or Un(*P1*)
F	T	Un	T	Un	**F**	**In**	Functions *P1* and *P2* achieved unexpectedly	max (Un(p1), Un(p2))
T	F	Fa	F	Fa	**F**	**Fa**	Function *F* failed because functions*P1* and *P2* failed	Fa(*F*)
T	T	Ac	F	Fa	**T**	**Ac**	Function *F* achieved because function*P1* achieved	
T	T	Ac	T	Ac	**F**	**Fa**	Function *F* failed because functions*P1* and *P2* achieved	Fa(*F*)

Table 7.3: Decomposition using partial incomplete functions with XOR

instead of PIFs, each subsidiary functions were complete functions each with its own (identical) trigger, then as the two triggers from which the triggering of the top level function will always share a value, the trigger of the top level function will always be false so the top level function can never be triggered and therefore never achieved correctly. This follows from the rule used that the truth of the triggering of the top level function is derived from the truth of the triggering of the subsidiary functions combined using the logical relation, so with `XOR` the top level function is only triggered if one of the subsidiary functions is triggered. This is shown in Table 6.4 on page 142. This difference does capture a real difference in the meaning of the functional decomposition, contrasting a function that is triggered by the trigger of one of its subsidiary functions with one that is triggered by (implicitly) both, even though only one of the effects is expected. This also means that partial incomplete functions are a necessary part of the functional language because this case cannot be described without their use.

An example of a system that might use such a functional description is not

easy to find as such a system uses a trigger with a non-deterministic effect. One possibility is a game where a random (or pseudo-random) element is necessary. A coin slot arcade game in which one of two model animals appears periodically for a short time, the idea being for the player to score as many hits as possible might be modelled in this way, in that a given trigger (either starting the game or, during play, an animal returning to its hole restarting the timer and selector) will result in either one but not both of the models appearing. This might be worth modelling using PIFs rather than simply effects as some play is possible if only one of the animals fails to appear. It is suggested that the somewhat contrived nature of this example serves to demonstrate how rarely `XOR` will be used in combining purposive incomplete functions.

In most cases, where `XOR` might be used to combine subsidiary functions, the trigger should be refined to eliminate this non-deterministic nature of the system. Main and emergency lighting systems might be considered to be PIF subfunctions of a light room function, either being triggered by switching on the lights and combined using `XOR` (or possibly `OR`). However, this fails to describe the fact that the emergency lighting system should only be triggered if the main system fails, so which system is triggered has a significance that is lost if `XOR` is used in describing these functions. In other words which effect is triggered is not non-deterministic and a correct description will have the emergency lighting function triggered by the state of the main lighting system, avoiding such non-determinism. Purposive incomplete functions are not appropriate in such cases, being intended principally to allow subfunctions that combine to achieve a function and that share a trigger to be described.

The discussion of relationships between states of purposive incomplete functions and the top level function is concluded by noting that the rule for including the consequences of failure of a function in the resulting design analysis report is, as might be expected, the same as when complete subsidiary functions are used. So when the PIFs are combined with AND or OR, the consequences are those of the failed PIF if one fails and the top level function if both PIFs fail. If `XOR` is used this is reversed (again as is the case with complete subsidiary functions) so if one PIF fails the top level consequences are included and if both do, the consequences of (one of) the PIFs. The unexpected achievement of PIFs is also handled in the same way as is unexpected achievement of complete subsidiary functions, so the consequences will depend as much on whether any have been added for unexpected achievement of function or whether those associated with individual effects are being relied upon.

7.2 Operational incomplete function

An operational incomplete function (OIF) defines a relationship between a trigger and effect that does not itself fulfil any purpose, but rather contributes to the fulfilment of the purpose of a function of which it is a subsidiary function. This might be felt to be inconsistent with the idea that any function is associated with a purpose and indeed an operational incomplete function is best regarded as a way of defining the expected behaviour of a system rather than its purpose. This raises the interesting point that a system function might be thought of as describing the system's expected behaviour as distinct from relating behaviour to purpose and in interpretation of simulation both of these might be of interest. This is discussed later, in Section 7.5. A description of an operational incomplete function therefore merely consists of a recogniser. An example of the use of operational incomplete functions might be a room with two independent lamp circuits, each with a switch and lamp. In this case, either will do to light an occupant's way around the room and arguably neither has a sufficiently distinct purpose of its own to justify the additional complication of completing their functional descriptions. The room light function might look like this.

```
FUNCTION room_light
  ACHIEVES find_way_around_room
  BY
    OIF wall_lamp
    OR
    OIF ceiling_lamp
```

Here, of course, the recogniser for the top level function is similar to cases where the subfunctions are complete functions. The only difference is that these subfunctions include no reference to a description of purpose.

```
OIF wall_lamp
  switch_on
  TRIGGERS
  light_on
```

As an operational subfunction consists of a recogniser, there is arguably no need to label it with the `BY` keyword. Because the subsidiary functions include both triggers and effects, the relationships between achievement of the top level function and the subsidiary functions is identical whether the subsidiary functions are complete functions or operational incomplete functions. They are therefore as illustrated in the tables in Section 6.3. Because there is no purpose attached

to an operational incomplete function, all consequences of failure are those of the top level function, just as though the top level function was described using triggers and effects. This means that the failure of one of a function's OIF subfunctions might not result in failure of the top level function, as in this example. For example, if both room lights are switched on but the wall light has failed, the `room_light` function is still achieved. This is, of course, no different from the case where a function consists of several effects. Consequences of unexpected achievement can either be those associated with the relevant effect or, if the top level function is achieved and it is associated with a set of consequences of unexpected achievement, with the top level function. As there is no purpose associated with an operational incomplete function, it will not have its own consequences of unexpected achievement. The rôle of an operational incomplete function is to ensure that a trigger is associated with the right effect. Notice that in this example, the wall light switch is associated with the wall light, while using `OR` to combine the triggers and effects in the `room_light` function loses this association.

```
FUNCTION room_light
  ACHIEVES find_way_around_room
  BY
    wall_lamp_switch_on OR ceiling_lamp_switch_on
    TRIGGERS
    wall_lamp_on OR ceiling_lamp_on
```

This is not equivalent to the use of the operational incomplete subfunctions as, of course, either switch could trigger either light, which is not a good model of the intended functionality. This could be avoided by having more than one `TRIGGERS` in the recogniser of the function, of course, like this.

```
FUNCTION room_light
  ACHIEVES find_way_around_room
  BY
    wall_lamp_switch_on
    TRIGGERS
    wall_lamp_on
    OR
    ceiling_lamp_switch_on
    TRIGGERS
    ceiling_lamp_on
```

It will be appreciated that these formulations give identical results if the trigger, effects or subfunctions are combined using `AND`.

Although the use of the formulation above avoids the necessity for using operational incomplete functions, there are arguments for using them. Their use allows

the idea that a function has one trigger and one effect (so `TRIGGERS` appears just once) to be retained. Another advantage of the use of operationally incomplete functions is that later in the design process, the design might become refined such that these subfunctions might acquire a distinct purpose of their own, so they can be promoted to complete functions. This is simply done by adding an `ACHIEVES` clause with a reference to the purpose to the description of the operational incomplete function. For example, once the layout of the room in the example above is known, it might become apparent that someone working at the desk will be working in their own shadow if the ceiling light is in use, so the wall light acquires a specific purpose, lighting work at the desk. If this functional refinement is done without the use of operational incomplete functions, a good deal more rebuilding of the functional models will be required.

The use of operational incomplete functions also allows a function to be decomposed such that only one of the subsidiary functions mitigates failure of the top level function. This problem was touched on in Section 6.2.1.1 earlier. To use the railway signalling example, the top level function might be written as follows.

```
FUNCTION accept_train
  ACHIEVES guide_to_platform
  BY
    OIF set_points
    AND
    FUNCTION set_signals
```

Now, if the points fail, the consequences of the main function will be included in the report, as there are no others available, while if the signals fail, those consequences are included, allowing the report to show that in this case, the loss of functionality in the system can be worked around by hand signalling. In other words, failure of the points amounts to complete failure of the system while failure of the signals is only a partial failure. While such asymmetric functional models will, it is suggested, be unusual it does seem worthwhile allowing the language the flexibility to describe such cases. There is a fuller discussion of this in Section 7.4.1. It will be seen that formulating the above example without using an OIF would be somewhat cumbersome.

As a final note on OIFs, their presence makes it necessary to refine the formal rule 5.13 in Section 5.7, as it is possible that a function has now associated description of purpose and so consequences of failure. To allow for this, the rule is changed so that consequences are only reported if the fucntion includes them, as follows.

$$\mathrm{R}(c_f) \text{ if } \mathrm{Fa}(f) \text{ and if } (f \text{ includes } c_f) \tag{7.5}$$

Where $\mathrm{R}(c_f)$ means the consequences of f are reported if f has failed and f inludes such consequences (in its associated description of purpose).

7.3 Triggered incomplete function

As was noted in the introduction to this chapter, this last class of incomplete function might be thought to be unnecessary, expressing as it does a relationship which requires an intermediary element (the effect). Indeed it is likely that triggered incomplete functions will be used a good deal less often than either of the other classes of incomplete function. The principal reason for its inclusion in the Functional Interpretation Language is completeness but there do exist systems whose description might make use of such an incomplete representation of function. A very simple example is a door with a tumbler lock (such as a “Yale” lock) where the releasing of the lock can be triggered either by using the key (with the purpose of letting somebody in) or the knob, with the purpose of letting somebody out. While there is arguably little need for such complication here, it will be noticed that the consequences of failure of these two purposes (and so triggers) is different and also that the consequences of failure of either one of the subsidiary functions is less severe than both. If, say, some fault renders the key unusable, someone wanting to get in could knock so that an occupant can open the door. If neither trigger works, the lock needs removing to open the door. This example might seem a little contrived, which does, perhaps, serve to suggest that this class of incomplete function will seldom be used. It does seem possible that it might be of some use in describing software systems, where the consequences of the output of a certain module might depend on how the module was called.

Formally, a function f composed of two TIFs s and u combined using some operator $\otimes$ has its function state defined following these rules.

$$\mathrm{In}(f) \Leftrightarrow \neg(\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \neg\mathrm{Ef}(f) \tag{7.6}$$

$$\mathrm{Fa}(f) \Leftrightarrow (\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \neg\mathrm{Ef}(f) \tag{7.7}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg(\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \mathrm{Ef}(f) \tag{7.8}$$

$$\mathrm{Ac}(f) \Leftrightarrow (\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \mathrm{Ef}(f) \tag{7.9}$$

These are similar to the rules for PIFs, above, but now the triggers are associated with the subsidiary functions and the effect with the top level function. A similar point regarding the use of `XOR` applies here as applies to PIFs in that if the same effect is associated with each subsidiary function, the top level function will never

be effective as both or neither effect will be true at any time.

The description of a triggered incomplete function is not unlike that for a purposive incomplete function but, of course, the TIF takes the place of the trigger expression, so the lock example might look like this.

```
FUNCTION unlock_door
  ACHIEVES door_open
  BY
    TIF unlock_with_key
    OR
    TIF unlock_with_knob
    TRIGGERS
    lock_released
```

Here, `lock_released` is an effect with two purposes, which are distinguished by the trigger used. The TIF's description consists of a reference to a description of purpose and a recogniser that is simply a trigger expression.

```
TIF unlock_with_key
  ACHIEVES let_in
  BY
    key_inserted_and_turned
```

The description of purpose (here `let_in`) is similar to one that might be associated with a complete function. It will not, generally, include a set of consequences of unexpected achievement of the effect. As there is no effect in the function, it would be inappropriate. If the effect (common to two TIFs) occurs unexpectedly it could apply to either or both of the TIFs. This allows the hierarchical functional description (and so the resulting design analysis report) to differentiate between the consequences of the effect failing in response to each trigger.

The relationships between states of triggered incomplete functions and the top level function are not dissimilar to those for purposive incomplete functions with the obvious difference that as the effect is shared, it is inherited from the top level function rather than the trigger. Combining triggered incomplete functions with `AND` has the results shown in Table 7.4. The results are similar to those lines in Table 6.2 on page 138 where both effects share a value. It will be appreciated that where `AND` is used, both triggers are required to trigger the expected effect. This leaves a limited rôle for combining triggered incomplete functions with `AND` as it will never be the case that one or other of the subsidiary functions will fail. However, it is possible that one of the TIFs is achieved unexpectedly when the

function *F* (*T1* AND *T2*)								generated text	consequences
T1		*T2*							
t1	state	*t2*	state	*tF*	*e*	state			
F	In	F	In	F	F	**In**		(no entry)	
F	Un	F	Un	F	T	**Un**		Function *F* achieved unexpectedly because effect *e* unexpectedly present	Un(*F*) or Un(*e*)
F	In	T	Fa	F	F	**In**		(no entry)	
F	Un	T	Ac	F	T	**Un**		Function *F* achieved unexpectedly because function *T1* achieved unexpectedly	Un(*T1*) or Un(*e*)
T	Fa	T	Fa	T	F	**Fa**		Function F failed because functions T1 and T2 failed	Fa(*F*)
T	Ac	T	Ac	T	T	**Ac**		Function *F* achieved	

Table 7.4: Functional decomposition using triggered incomplete functions with AND

effect appears in response to one of the two triggers, so the other TIF is achieved unexpectedly. There is therefore the possibility of using TIFs combined with AND where it is required to add consequences of unexpected achievement of a function's effects at this intermediate level in the functional hierarchy. It is therefore not impossible that the description of purpose associated with a TIF will include a set of consequences of unexpected achievement.

There is a more definite rôle for triggered incomplete functions combined with OR, indeed the lock example mentioned in the introduction to this section is a case in point. Here, again, the relationships between the TIFs and the achievement of the top level function, shown in Table 7.5, are similar to those for complete subsidiary functions, compare with table 6.3 an page 140. If one of the TIFs fails, the consequences are those of the TIF while if both fail (neither trigger will result in the expected effect) the consequences are those of the top level function. This is consistent with the rule used with complete subsidiary functions and purposive incomplete functions, and is also used where TIFs are combined with AND. This leads to an interesting side effect with the lock example, as in this case it is unlikely that both triggers will occur simultaneously. Therefore the interpretation must allow for this and compare the results of the use of each trigger to establish that neither results in the expected effect and therefore that the top level function's consequences are to be included in the report. There seems to be no problem with this, provided the results of the simulation(s) allow this comparison to be made.

As is the case with purposive incomplete functions, the relationship between subsidiary triggered incomplete functions and the top level function using XOR is different from that of using two complete subsidiary functions with the same effect.

function *F* (*T1* OR *T2*)							generated text	consequences
T1		*T2*						
t1	state	*t2*	state	*tF*	*e*	state		
F	In	F	In	F	F	**In**	(no entry)	
F	Un	F	Un	F	T	**Uc**	Function *F* achieved unexpectedly because effect *e* unexpectedly present	Un(*F*) or Un(*e*)
F	In	T	Fa	T	F	**Fa**	Function *F* failed because function *T2* failed	Fa(*T2*)
F	Un	T	Ac	T	T	**Ac**	Function *F* achieved	
T	Fa	T	Fa	T	F	**Fa**	Function *F* failed because functions *T1* and *T2* failed	Fa(*F*)
T	Ac	T	Ac	T	T	**Ac**	Function *F* achieved	

Table 7.5: Functional decomposition using triggered incomplete functions with OR

These relationships are shown in Table 7.6. Here an effect is triggered by one or

function *F* (*T1* XOR *T2*)							generated text	consequences
T1		*T2*						
t1	state	*t2*	state	*tF*	*e*	state		
F	In	F	In	F	F	**In**	(no entry)	
F	Un	F	Un	F	T	**Un**	Function *F* achieved unexpectedly because effect *e* unexpectedly present	Un(*F*) or Un(*e*)
F	In	T	Fa	T	F	**Fa**	Function F failed because function T2 failed	Fa(*F*)
F	Un	T	Ac	T	T	**Ac**	Function *F* achieved	
T	Fa	T	Fa	F	F	**In**	(no entry)	
T	Ac	T	Ac	F	T	**Un**	Function *F* achieved unexpectedly because effect *e* unexpectedly present	Un(*F*) or Un(*e*)

Table 7.6: Functional decomposition using triggered incomplete functions with XOR

other of two alternative triggers, rather then a top level function being achieved by one or other of two subsidiary functions. If complete subsidiary functions are used in place of the triggered incomplete functions, then if one subsidiary function's effect is present, so is the other. The two effects combined with `XOR` will, therefore always resolve to false, so the top level function would never be achieved. This is consistent with the rule used for functional decomposition, where the truth of a top level function's effect is derived from the truth of the subsidiary functions' effects combined using the relevant logical operator. A possible example of using TIFs with `XOR` is the common arrangement for a landing light with two switches, and the light lighting when either one of the switches in on. This might be described

like this.

```
FUNCTION landing_light
  ACHIEVES light_stairwell
  BY
    TIF downstairs_switch
    XOR
    TIF upstairs_switch
    TRIGGERS
    lamp_on
```

It is at least arguably the case that the purposes of these two TIFs are not really distinct, but it is possible that the functions are described this way to allow for the idea that one of the switches failing to switch on the lamp is a less serious failure of the system than both failing. This works because each TIF will include a related description of purpose besides the expression for the trigger, like this.

```
TIF upstairs_switch
  ACHIEVES light_way_down_stairs
  BY
    upstairs_switch_pressed_down
```

As TIFs combined using `XOR` cannot be replaced with complete subsidiary functions, because the resulting relationship with the top level function is different, TIFs are a necessary component of the functional language, even though it is likely they will only infrequently be used.

Where PIFs or TIFs are used in a functional decomposition, the rules for reporting failures of functions follow those for complete subsidiary functions, A.11 and A.12 in Appendix A and the rules for reporting the consequences of these failures follow A.18, A.19 and A.20.

7.4 Incomplete functions in functional decomposition

The three classes of incomplete function are used as subsidiary functions of a complete function to assist with the construction of a hierarchical functional decomposition. A function can therefore be decomposed in five ways, using triggers and effect, complete subsidiary functions, or any class of incomplete function, as illustrated in figure 7.1. In the figure, a functional description is shown by a thick

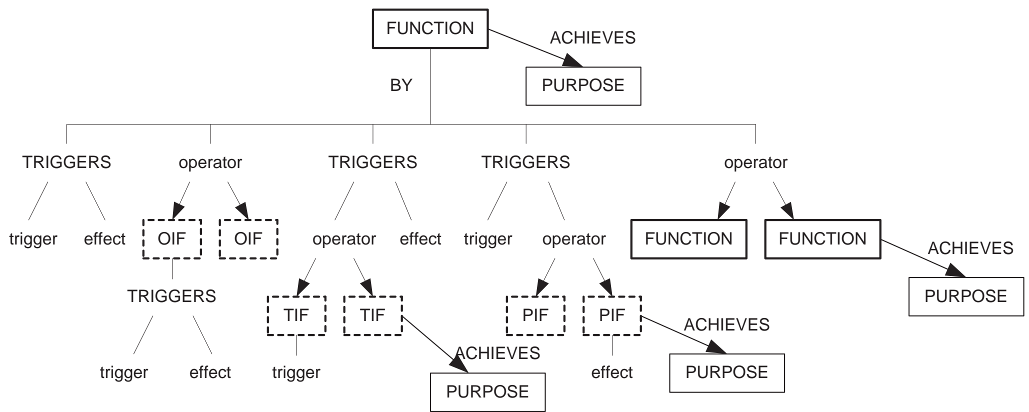


Figure 7.1: Five functional decompositions

lined box while a thin box used for an associated description of purpose. Arrow heads are used to distinguish relationships between model components that are (or could be) seen as separate models, in different files. Incomplete functions are shown by a broken outline and "operator" is used to indicate any Boolean operator. Any "trigger" and "effect" in the diagram can, of course, be a Boolean expression. There is a key to this and other similar figures in Appendix C on page 286.

Where an effect is associated with a distinct purpose (so the consequences of its failure are different from those of failure of the top level function), then either a complete function or a purposive incomplete function is used, depending on whether the effect is associated with its own trigger. Otherwise, the function is simply composed either of trigger and effect or, if there is more than one of each and they need associating, operational incomplete functions are used.

It is possible (though perhaps unlikely) that these decompositions might be mixed, though it will be apparent that there are restrictions on this mixing. It will be seen in figure 7.1 that three of the decompositions start with `TRIGGERS` and the other two with a Boolean operator. It is possible to mix elements from these related decompositions, so a purposive incomplete function can be paired with an effect that fulfils no independent purpose, a complete subsidiary function can be paired with an operational incomplete function and a triggered incomplete function with a trigger expression that triggers the same effect. This might be done in cases where a function depends on two effects, one of which has a clearly defined purpose (or clearly defined consequences of failure) of its own while the other does not. This allows functional decompositions that are not symmetrical, in that one of the effects is more essential to correct achievement of a function than the other.

7.4.1 Asymmetrical functional decompositions

An advantage of the approach to functional decomposition described here is that it allows functions that might be considered to be asymmetrical to be described. What is meant by this are the cases where a function depends on two effects (say) only one of which can be regarded as a subsidiary function or where a function depends on two subsidiary functions only one of which is regarded as mitigating the failure of the top level function. While such cases are likely to be unusual, it not impossible that a trigger has effects of different degrees of significance in achieving a function. A simple example might be a car's brake lighting function, where the car has a repeater in addition to the two required brake lights. As the brake lights proper are legal requirements it could be argued that their failure is more significant than failure of the repeater. This might be described by mixing effects and a purposive incomplete function, like this.

```
FUNCTION stop_lights
  ACHIEVES warn_of_slowing
  BY
    press_brake_pedal
    TRIGGERS
    left_brake_light
    AND
    right_brake_light
    AND
    PIF stop_light_repeater
```

The purposive incomplete function will, of course, have its own description of purpose and effect.

```
PIF stop_light_repeater
  ACHIEVES additional_slowing_warning
  BY
    repeater_lights

PURPOSE additional_slowing_warning
  DESCRIPTION "More visible warning of slowing vehicle"
  FAILURE "No extra warning of slowing"
    SEVERITY 3
    DETECTION 6
```

It will be seen that in this case, failure of either brake light is treated as failure of the top level function, while failure of the repeater is less significant, as the consequences of its failure will be those of the failure of the purposive incomplete function, presumed to be less severe than those of the main function.

The easiest ways of describing these asymmetric decompositions are by combining effects and purposive incomplete functions and by combining complete subsidiary functions and operational incomplete functions. This was illustrated in Section 7.2. It is also possible to combine triggered incomplete functions and trigger expressions in the same way. While it will perhaps be an unnecessary refinement in the functional description for interpretation of simulation, there might conceivably be cases where it is of use in functional refinement of a design, by helping to identify those effects that are seen as more essential to achieving some system function and so possibly focussing the provision of additional fault tolerant functionality.

7.5 Differences between behaviour and purpose

The use of incomplete functions to represent the relationship between trigger and effect and effect and purpose suggests that the complete representation of a function actually captures both these different, if related, relationships. This raises the possibility that the relationship between the trigger and effect, the system's expected behaviour, does not require the same effect as the purposive relationship. The most typical case of this is where a trigger triggers redundant effects. A possible example where these two aspects of a function do not require the same relationship between effects is where a room has two lights that share a switch. The expected behaviour is clearly

```
switch_on
TRIGGERS
ceiling_light_on AND wall_light_on
```

However, it might be felt that either one of these lights being lit is sufficient to fulfil the purpose of letting occupants find their way around the room. The significance of this consideration being, of course, that a system failure that results on the loss of one light is less severe than one that results in the loss of both. Ignoring the trigger, then the purposive relationship might be written as

```
ACHIEVES light_way_around_room
BY
  ceiling_light_on OR wall_light_on
```

This raises the question of whether in fact a complete function should be split into a purposive and an operational component to capture this difference. While such an approach is interesting, it is rejected because of the additional complexity

required to capture what might well be regarded as a somewhat pedantic distinction (in that the functional description could readily be built around the expected behaviour) and the relative rarity of such cases.

If it is felt necessary to include this distinction in the model, subsidiary functions can be used by combining them with `OR` but having them share a trigger, so that one should never be achieved without the other, like this.

```
FUNCTION room_light
  ACHIEVES light_way_around_room
  BY
    FUNCTION wall_lamp
    OR
    FUNCTION ceiling_lamp

FUNCTION wall_lamp
  ACHIEVES light_desk
  BY
    switch_on
    TRIGGERS
    wall_lamp_lit

FUNCTION ceiling_lamp
  ACHIEVES light_room
  BY
    switch_on
    TRIGGERS
    ceiling_lamp_lit
```

For both subsidiary functions to share a trigger, both `switch_on` labels need attaching to the same system property, of course. This being done neither subfunction should be achieved without the other, but either can be regarded as achieving the `room_light` function. If the `wall_lamp` function fails, for example, the report will include

> On switching the lights on, function wall_lamp not achieved.

The consequences and severity will, of course, be those associated with that lower level function.

In practice, similar results in the resulting report can be achieved more simply using purposive incomplete functions. These can be combined using `AND`, to represent the expected behaviour, or `OR`, to represent the purposive relationship, as the rule used for reporting on the failure of one subsidiary function means that the results will be not dissimilar in both cases. As was discussed in the previous

chapter, the only difference is the loss of the reference to the top level function if OR is used. It is suggested that these approaches are sufficient for adequate description of cases where the expected behaviour differs from the representation of the purposive relationship and so there is little or no need to express these two relationships distinctly in the functional representation.

So far the basis for the present Functional Interpretation Language has been described, together with how system functions can be decomposed either into expected effects or into subsidiary functions, which need not themselves be complete representations of function. There are other dimensions to functional decomposition that might be required however. These are the requirement to specify a temporal aspect to functional elements, either in terms of a function depending on a sequence of subsidiary functions or effects or to allow the untimely (typically late) achievement of a function to be described and also to allow functions whose state depends on the achievement of some other system function to be modelled. The following chapters examine how the present language addresses these needs, and describes and discusses the extensions required to enable such cases to be modelled.

Chapter 8

Describing functions that depend on terminating, intermittent and sequential behaviour

The approach to functional decomposition described in the preceding chapters is adequate for describing system functions where a trigger results in some effect (whether output or goal state) that then lasts until some other trigger results in some further change in the state of the system. For example a lamp that, having been switched on, remains lit until it is switched off again. However, there are many systems whose functionality depends on some terminating or intermittent change of state, or a sequence of such changes, that cannot readily be described using the conventional logical operators. A simple example is the confirmation of remote locking by two flashes of all of a car's direction indicators where the effect of the confirmation function is completed and the system returns to a similar state to that before the triggering of that function with no further external stimulus.

To describe functions that depend on such behaviour, some way of expressing temporal relationships between successive effects is required. The conventional logical operators lack any temporal element. For example, in defining a function's effects (or subfunctions) using `AND`, all that is stated is that the two effects should both be present at some indeterminate time. This is appropriate for describing persistent effects (such as a light staying on) as the results of the simulation can be checked against the functional model at some suitable point in time. A suitable point is once the system's response to a triggering event (such as the simulated throwing of a switch) is behaviourally complete and the system is in a steady state awaiting the next stimulus. The simulation of a system's response to a triggering event is therefore implicitly placed in a single time slot, beginning with the trigger

and ending once the system is in a steady state, whereupon the effects (outputs or goal states) are compared to the system's functional model to establish what functions are achieved and what effects are unexpected. This is the approach taken by the design analysis tool developed in earlier work at Aberystwyth and is illustrated in Figure 8.1. In this and similar diagrams, the horizontal axis

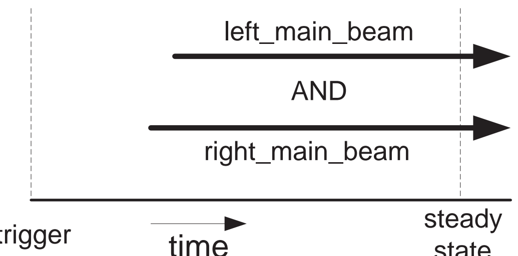


Figure 8.1: Using logical operators resulting in a single time slot

represents time and the vertical axis the decomposition of the function's effects (or subsidiary functions). A thick line represents the presence of the named effect and an arrow head its continuation beyond the time represented in the diagram, as in this case where the main beam effects continue indefinitely. Here, the effect is checked against the functional model at the end of the simulation step, taken to be once the simulated behaviour of the system reaches a steady state. As the expected effects are present at that point in time, the function is achieved, but it will be seen that strictly speaking the model fails to specify that the two effects start together and the possibility that they do not is ignored. This is unlikely to be a problem in most cases. Typically the effects will start together but if they do not, it does not affect the intended functionality of the system. For example, it is possible (if unlikely) that the two headlamps in the example have their current switched by different relays and that a fault in one of them delays that headlamp switching to main beam so both effect do not start simultaneously. If the state of the system is only checked at the end o the time slot, this will be missed. This case does serve to illustrate the fact that the conventional logical operators lack any temporal dimension. This approach is, however, clearly inappropriate for modelling systems whose functions depend on effects that depend on some behaviour that is completed before the system settles into a steady state. For example, in the remote locking confirmation function mentioned earlier, the flashes will be over before the system enters a steady state, so these effects will be missed if comparison with the functional model is only made then. Consider a simple warning system that, for simplicity, flashes a lamp and sounds a horn in response to some stimulus, as illustrated in Figure 8.2. This simple system is used as

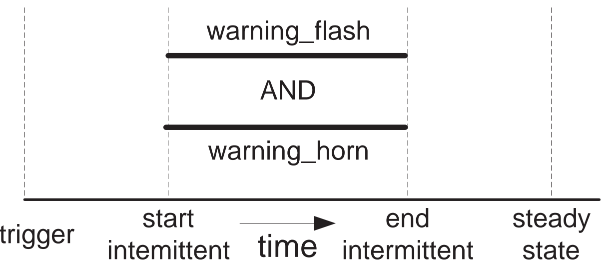


Figure 8.2: Finding function at the end of simulation might miss significant effects

an illustration during this chapter, before discussing the modelling of sequential behaviour with a more realistic case study. It will be seen that if these terminating effects (which might, of course, be combined to create a sequence of several outputs such as a number of repeated flashes) are not to be lost in comparison with the functional description, what is required are operators that allow the ordering of such effects to be specified and that also indicate the intermediate time steps which require comparison with the functional model to ensure any terminating effects that are found in running the simulation of the system are compared with those specified by the functional description.

To meet this need, the functional language incorporates two sequential operators similar to operators found in temporal logic. These are

- `SEQ` (or "strict sequence") resolves to true if the following condition resolves to true in the next time slot so the following state immediately succeeds the preceding system state. This is similar to the **N** "in the next time step" operator found in some temporal logics, such as Computational Tree Logic, abbreviated to CTL (Emerson & Halpern, 1985).
- `L-SEQ` (or "loose sequence") resolves to true if the following condition is true some time after the operator's place in the temporal sequence, similar to the tense logic **F** "some time in the future" operator (Prior, 1957).

The use of a next time step operator entails the use of a model of time that consists of intervals that meet such that one can be considered to follow immediately after another. The interval model of time in (Allen, 1984) is such a model as while he does not regard an interval as containing instants, he does have a relation MEETS(a, b) that is satisfied if interval b follows interval a with no intervening interval. In other words, (Allen, 1984) does not explicitly model instants, but that relation implies the possibility of instantaneous transitions between intervals. The notion of "properties" that can hold (or not hold) during an interval can be likened

to system states that are true during an interval. There is a relation HOLDS(p, t) that is satisfied if the property p is true in the interval t and its sub-intervals. To answer difficulties found in modelling continuous change using this model of time, it was refined in (Galton, 1990). In that model of time, intervals can contain instants so, for example, the interval during which something is travelling North can contain the instant when that object passes due West of some other object. Allen's HOLDS relation is subdivided into three relations.

- HOLDS-AT(p, I) is satisfied if p is true at instant I.
- HOLDS-IN(p, T) is satisfied if p is true at some instant within interval T. Therefore p need not be true throughout interval T.
- HOLDS-ON(p, T) is satisfied if p is true at all instants within interval T, so it is true throughout T.

As the Functional Interpretation Language was devised for functional modelling of systems whose behaviour was expressible using state charts, the `SEQ` can `L-SEQ` operators can be expressed in terms of intervals and properties. The Functional Interpretation Language sequential operators can be defined in terms of the MEETS relation in (Allen, 1984) and the HOLDS relations from (Galton, 1990), as follows. Let a and b be some specified system states (that are the required effects of some function) and I_1 and I_2 are intervals of time. The relation a SEQ b is true if an interval throughout which a is true is immediately followed by an interval throughout which b is true.

$$a \text{ SEQ } b \text{ if } \text{HOLDS_AT}(a, I_1) \wedge \text{HOLDS_AT}(b, I_2) \wedge \text{MEETS}(I_1, I_2) \tag{8.1}$$

The relation a L-SEQ b is true if an interval in which a is true at some point is immediately followed by an interval in which b is true at some point.

$$a \text{ L-SEQ } b \text{ if } \text{HOLDS_IN}(a, I_1) \wedge \text{HOLDS_IN}(b, I_2) \wedge \text{MEETS}(I_1, I_2) \tag{8.2}$$

As is the case in temporal logics such as CTL (Emerson & Halpern, 1985), the sequential operators (both `SEQ` and `L-SEQ`) are unary so they resolve to true provided the succeeding state resolves to true (in an appropriate time step). If the preceding state is not achieved, the sequence has already failed, of course. This means that a sequence can begin with the operator, referring to the first effects triggered by the relevant triggering event, as here. This is illustrated in the section discussing the use of `SEQ`.

One question that arises from the similarity of these sequential operators to the "future" operators from temporal logic is whether there is any need for equivalents of the past operators. As the aim of these operators is to allow the functional description to describe a sequence of effects that can be matched by tracing forwards through a description of the sequence of states of the behavioural simulation, it is suggested that such operators are not necessary. This might be likened to their absence in CTL (Emerson & Halpern, 1985) which also traces forward through a graph of possible states of a system. One area where past operators might be used is in adding the effect of an earlier trigger, such as an over-ride of some function whose effects are permanent, or at least beyond the present use of the system. This is a feature of the seat belt warning system discussed in Chapter 11 where if the user buckles and unbuckles the driver's seat belt five times in succession, the system is permanently disabled and no warning will be given in future. Rather than using "past" operators in the trigger of each function, it is possible to treat this behaviour as an additional function of the system and use the goal state of this function as the condition for the trigger of the warning function. The use of functions in the triggers of other functions is discussed in Chapter 10.

The use of the sequential operators is discussed in the following sections before considering how the language models sequences of effects that continue indefinitely and illustrating the use of these features of the language with simple case studies.

8.1 The strict sequence operator

As noted above, the operator `SEQ` is used to specify changes that should immediately follow one another. It can therefore be used to specify that the flash and horn of the simple example warning system should be simultaneous, as illustrated in Figure 8.3. If the flash and horn are treated as required effects of a warning function, the functional description itself can be written as follows.

```
FUNCTION warning
  ACHIEVES warn_user
  BY
    malfunction
    TRIGGERS
      SEQ (warning_flash AND warning_horn)
      SEQ (NOT warning_flash AND NOT warning_horn)
```

In this case, NOT is used to show that the effects should be absent during certain intervals in the sequence. This avoids the need to associate any function with the

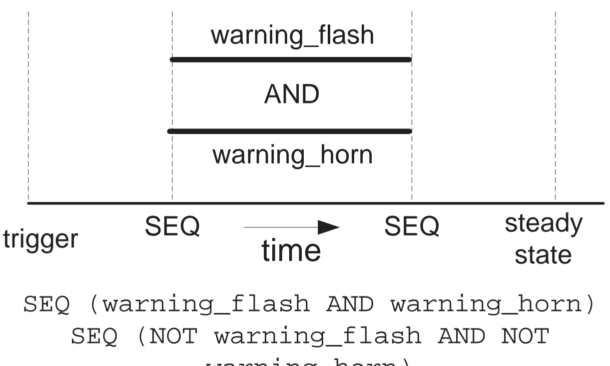


Figure 8.3: Specifying sequences of immediately successive effects

absence of the effects. In this case this is acceptable as we can regard the effects as being either "on" or "off", the lamp is either lit or not lit for example. The use of NOT is discussed in subsection 8.1.2 below. The fact that the sequential operators are unary means that they can be used at the start of a sequence as here. This will frequently be the case as the initial state of the system will be defined in some other functional description. In this case, `SEQ` is satisfied if the first state transition following the trigger that involves any change in the effects included in the functional model results in the state following `SEQ` being true. This allows `SEQ` to be used to specify that two effects are achieved simultaneously in cases where a single time slot is otherwise sufficient, such as shown in Figure 8.1 above. For example, if it is felt necessary to specify that a car headlamps' main beams come on simultaneously, the functional description might be written using `SEQ`.

```
FUNCTION main_beam
  ACHIEVES light_road_ahead
  BY
    lamp_switch_heads AND dip_switch_main
    TRIGGERS
    SEQ (left_main_beam AND right_main_beam)
```

The fact that this specifies that a time step in which the headlamps are not on main beam (implicitly) is immediately followed by one in which they are both on main beam stipulates that the lamps come on together. The single time slot is divided into two, the second of which must immediately follow the first as illustrated in Figure 8.4. The the system states, and so the effects present, at the beginning of a simulation step are unknown as that depends on the previous trigger. In the headlamps example, the main beam function might either be triggered by the dip switch being switched to the main beam position or by the headlamps being switched on, if the dip switch was left in that position, so they

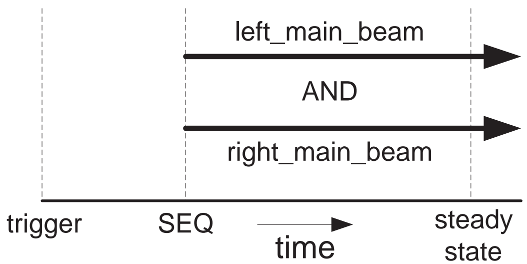


Figure 8.4: Using SEQ to specify effects that should start together.

first come on in main beam. Either of those triggering operations could therefore make the function's trigger true. The state of the headlamps at the time of the triggering event is therefore unknown and so not part of this functional description.

The use of this operator both specifies the expected behaviour that achieves the function concerned and indicates that the simulation step will have to be divided into successive time steps, beginning with the triggering event and ending with the system entering a steady state ready for the next triggering event. The intermediate time steps are delineated by the occurrences of `SEQ` in the functional description. This means that for interpretation using these sequential operators, the simulation engine must provide a sequential description of the system states associated with the simulation step rather than simply a description of the state of the system at the end of the simulation. This means that the interpreter can step through the description checking the state of the system at each step. The step in the simulation preceding the first instance of `SEQ` and, especially, the step following the final instance might be of zero duration as the system immediately enters a steady state. The system might well enter a steady state immediately following the ending of the flash and sounding of the horn in the simple warning example.

The succession of system states specified by the functional description are, of course, purely concerned with changes to the system's effects, rather than any internal system states. For example, suppose the headlamp system in the example above uses a relay to allow a relatively low current in the dashboard switch to switch the higher current needed to light the headlamps. In this case, the simulation will start with the dashboard switch being changed, which will result in current flowing through the relay's coil. This will, in turn, cause the relay switch to trip and allow current to flow through the lamps, resulting in the achievement of the function. This results in an intermediate state in which current is flowing through the coil, before the lamps light. As this change of state is purely internal it does not affect the functional model. What is of interest for interpretation is

changes of system behaviour that changes the effector components, those that implement some part of a function's expected effects. The simulation side of a design analysis tool will provide a complete step by step description of the results of the simulation and the interpreter will ignore those steps that do not involve a change in the effects generated by the system. This means, to return to the example, that the intermediate state of current flowing in the relay coil does not affect the truth of the SEQ operator as neither headlamp changes state. The behavioural simulation can be expected to list all these changes of state (whether the effects are internal or external) so the sequential functional description indicates which of these changes are functionally significant. This is illustrated in the example simulation in Chapter 11.

While `SEQ` will perhaps be most commonly be used to describe the succession of effects required when a function depends on intermittent or sequential behaviour, as discussed in (Bell & Snooke, 2004), there is no reason why the operator should not be used to specify a sequence of subsidiary functions. Indeed, the simple warning system might well have the horn and lamp associated with their own purposive incomplete functions on the grounds that achievement of either effect mitigates the failure of the warning function. As purposive incomplete functions are used in place of effects in a functional description, the resulting functional description might look like this.

```
FUNCTION warning
  ACHIEVES warn_user
  BY
    malfunction
    TRIGGERS
      SEQ (PIF visual_warning AND PIF audible_warning)
      SEQ (NOT PIF visual_warning AND NOT PIF audible_warning)
```

The subsidiary functions are then associated with their own descriptions of purpose and effects as described in the previous chapter. The operators might also be used to specify a sequence of triggering events required to achieve a function, such as pressing and releasing a push switch that toggles a functional change. Notice that the subsidiary functions need to be associated with the presence of the required effect Consistently with the example above, as shown below.

```
PIF visual_warning
  ACHIEVES warn_user_visually
  BY
    show_warning_lamp
```

The sequence of subsidiary functions (and so their effects) must be managed by the top level function to specify the temporal relationship between the two effects. If the sequence of lamp on and lamp off is specified in the subsidiary function (and likewise for the horn) then the functional description cannot specify that the lamp and horn start and end simultaneously. To illustrate, if the sequences in the subsidiary PIFs, like this.

```
PIF visual_warning
  ACHIEVES warn_user_visually
  BY
    show_warning_lamp SEQ NOT show_warning_lamp
```

The audible warning PIF will be similar. The PIFs are then related using AND.

```
FUNCTION warning
  ACHIEVES warn_user
  BY
    malfunction
    TRIGGERS
      visual_warning AND audible_warning
```

This functional model fails to specify any relationship between timing of the flash of the lamp and the sounding of the horn, as they are combined using AND. This is discussed further in Section 8.2. One difficulty this gives rise to is that the subsidiary function (and so the subsidiary purpose) is associated only with the "active" effect, not with the expected sequence of effects. In this case, that does not seem particularly problematic as it is the lighting of the warning lamp and sounding of the horn that give the warning.

The use of the SEQ operator entails a discrete model of time where intervals are divided by instantaneous "points" in time when a change might occur. This is not dissimilar to the model of time in (Galton, 1990). This model of time is more appropriate in some application fields than others, of course, and is usable where a behaviour can be described using a state transition diagram. Strictly speaking in many systems changes of state will not be instantaneous, such as where a mechanical system accelerates to its operating speed, overcoming the inertia of its resting components. This difficulty could be avoided either by specifying intermediate states or by using the loose sequence operator, described below. In many cases it would be quite appropriate to simply ignore the intermediate states, so an electric motor might be modelled as either stopped (no current) or running if there is current in the windings. The fact that when it starts running it is

accelerating to overcome inertia in the mechanical part of the system can be treated as insignificant. Including intermediate states in a functional description is arguably an unnecessary complication, especially as these states do not, typically, relate to any purpose (other than reaching the final state). The only reason for a washing machine, say, having a "filling" state is to manage the transition between empty (so the door can be opened or spin drying can take place) and full (for washing or rinsing).

8.1.1 The sequence operator and temporal orderings

Because the strict sequence operator relies on a similar interval based ontology of time to that in (Allen, 1984) the operator can be used to define the temporal relations defined in that paper. There are thirteen of these relations, listed in Table 8.1. Twelve of these actually pair off into six pairs. The table also shows how SEQ can be used to describe these relations. It will be seen that using the

Relation	Meaning	Description
x before *y* *y* after *x*		(NOT *x* AND NOT *y*) SEQ (*x* AND NOT *y*) SEQ (NOT *x* AND NOT *y*) SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x meets *y* *y* met-by *x*		(NOT *x* AND NOT *y*) SEQ (*x* AND NOT *y*) SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x overlaps *y* *y* overlapped-by *x*		(NOT *x* AND NOT *y*) SEQ (*x* AND NOT *y*) SEQ (*x* AND *y*) SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x during *y* *y* contains *x*		(NOT *x* AND NOT *y*) SEQ (NOT *x* AND *y*) SEQ (*x* AND *y*) SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x starts *y* *y* started-by *x*		(NOT *x* AND NOT *y*) SEQ (*x* AND *y*) SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x finishes *y* *y* finished-by *x*		(NOT *x* AND NOT *y*) SEQ (NOT *x* AND *y*) SEQ (*x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x equal *y*		(NOT *x* AND NOT *y*) SEQ (*x* AND *y*) SEQ (NOT *x* AND NOT *y*)

Table 8.1: Using SEQ to describe different temporal relations

strict sequence operator allows the correct ordering of the starts and ends of each effect to be specified. In describing these relationships, the initial state has been included to avoid any possible suggestion of ambiguity. The terms for the relations are actually taken from (Gerevini & Schubert, 1995) but the notion of causality implicit in these terms is not appropriate in this case. For example, the relation in which both effects start together and x finishes before y is called "x starts y" which suggests that x is the cause of y. This will not be the case when describing a function's effects as they will share a trigger. Where functional modelling is used for interpretation of simulation, there is no need for any notion of causality

in the functional description, beyond the notion of a trigger. A full causal chain (or net) can be derived from the behavioural model of the system if it is required. The causal aspects of the relationships in (Gerevini & Schubert, 1995) are what distinguish the individual elements in the paired relationships.

It will be seen from Table 8.1 that `SEQ` can be used to describe any of these thirteen temporal relations when combined with `AND` and `NOT`, though this summary does depend on the effects being of a binary (on or off) nature. The use of `NOT` is discussed further below. It will also be appreciated that while sequences of effects have been kept short here, for simplicity and clarity, there is no reason why they should not be of arbitrary length, described simply by including repeated states linked using the sequence operator (or the loose sequence operator).

8.1.2 Using logical NOT

In the examples used above, logical `NOT` has been used to specify the absence of a particular effect, specifically (in the example) that once the simulation step finishes, neither the warning lamp should be lit nor the horn sounding. This is appropriate in this case as each effect can be regarded as being binary in nature - the lamp is either lit or not lit and the horn either sounding or silent. These two alternative states can be associated with the presence or absence of current, if the simulation is to be qualitative, or with suitable ranges of current if the simulation is numerical. This use of `NOT` saves any need to specify an "off" effect. Many effectors can be described in these terms (such as a flashing lamp alternating between on and off) but others cannot be so described. For example a windscreen wiper system might have two speeds, slow and fast, and an intermittent wipe feature that alternates between slow and stopped. Clearly in this case `NOT slow` cannot be used as having the intermittent wipe alternate between slow and fast would fit that description. Therefore it might well be the case that a "null" effect in which nothing is done might be necessary. The use of such a null effect could be avoided by use of `NOT`, of course, by combining all the available effects, so the windscreen wipers being stopped can be written as `NOT slow AND NOT fast` but this will become decidedly cumbersome in cases where an effector has several effective states.

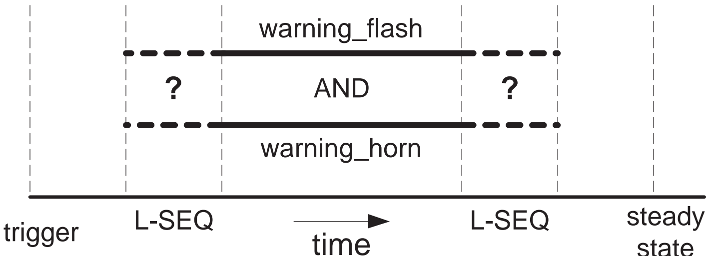
The description of functions that depend on an intermittent behaviour that alternates between an active and passive state (such a lamp flashing) is perhaps the most likely reason for using `NOT` in functional descriptions. Another possible use (which has less to do with describing intermittent behaviour) is if it is desired to include the absence of a specific behaviour that is inimical to achievement of a

function. For example, as the purpose of dipped headlamps is to light the road without dazzling the driver of oncoming traffic, if either headlamp's main beam filament remains lit, the purpose is not achieved (as drivers will be dazzled). It might therefore be considered appropriate to include the absence of such inimical effects in the functional description. The headlamps dipped effects therefore reading `left dipped AND right dipped AND NOT left main AND NOT right main`. This is generally unnecessary because the main beams being lit will be included in the design analysis report as unexpected effects. Relying on these inimical effects being marked as unexpected also has the advantage of distinguishing between the consequences of failure of a function because expected effects are missing and failure because unexpected effects are present. Therefore such complication of a functional description is seldom beneficial. One case where it might be useful is where a function has a likely side effect that must be avoided and whose effect is not covered by any other function, so will not otherwise appear in the functional model.

8.2 The loose sequence operator

The strict sequence operator described above is appropriate for describing intermittent behaviours that can be described as switching immediately from one state to another, using an interval ontology of time, but is not useful for describing changes that entail an intermediate state that is not part of the functional model. For example a washing machine's full state cannot immediately follow its empty state as the filling operation will not take place in an instant. There is an intermediate "filling" state that is an inevitable part of the behaviour of the machine. The transitions between empty, filling and full could be specified using `SEQ`. However, it is arguable that the filling state need not be specified in the functional model. The only effect of that state is internal in that its goal is for the machine to be full. These cases can be described using the "loose sequence" operator, `L-SEQ`. An expression using this operator resolves to true if the succeeding expression resolves to true at some time in the future. Therefore, there can be intermediate states that need not be specified in the functional description, such as the filling state in the washing machine example. This increases the flexibility of the language, the model builder can decide whether or not these intermediate states should be included in the functional description.

This operator allows two effects (or subsidiary functions) to be temporally related in a less constrained manner than `SEQ` allows, as illustrated in Figure 8.5. In this case, the functional description is satisfied provided both lamp and horn are



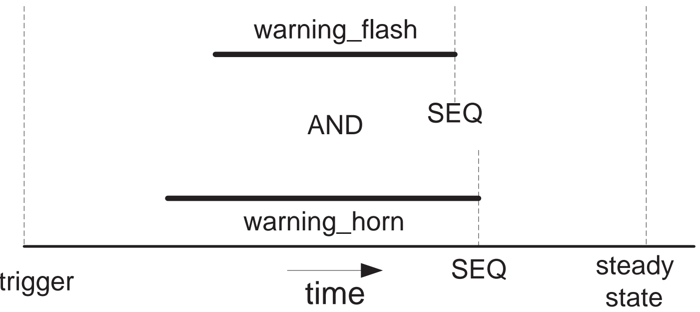

```
L-SEQ (warning_flash AND warning_horn)
L-SEQ (NOT warning_flash AND NOT warning_horn)
```

Figure 8.5: The loose sequence operator allows less closely constrained temporal relationships

active at the same time at some period of the simulation, but they need not start and end simultaneously because the period where one or the other is active is covered by the undefined intervals between both effects being active and both being passive. The use of `L-SEQ` avoids the need to specify intervals where one or other of the effects is present, though this could be done using `SEQ` and `OR`. This complicates the functional description and where effects are binary ("on" or "off") in nature this complication can be avoided. A possible example is a system that requires two valves to open to allow flow between two reservoirs. In this case, it is likely to be unimportant which valve opens or closes first, provided there is a period in which both are open, so the flow is possible. This allows a further set of less closely constrained temporal relationships to be described, as shown in Table 8.2. This set of partly ordered relationships can be described using a combination of `SEQ` and `L-SEQ`, where `L-SEQ` is used to describe those aspects of the relationship where the order need not be specified, provided that there is no danger of unwanted intermediate states being missed. The only exception is the case where two terminating effects are temporally unrelated, where two separate sequences are combined using AND. This means that both effects must have been achieved at some point during the simulation but the ordering is unimportant. Either can occur before the other or they can occur together, in which case they need not start or end simultaneously. This can be likened to the discussion at the beginning of this chapter, in which it was noted that `AND` has no temporal aspect, but complicated by the presence of two sequences of effects. This allows branching of time to be represented, as in such temporal logics as CTL, where it is not the case that some event must always be before, after or simultaneous with some other event. This case is illustrated in Figure 8.6. While this is illustrated using `SEQ`, `L-SEQ` could be used where possible intermediate states are to be allowed for.

Relation	Meaning	Interpretation	Description
x unordered-with *y*	*x* ? *y*	both *x* and *y* must occur at some stage in the simulation step	(NOT *x* SEQ *x* SEQ NOT *x*) AND (NOT *y* SEQ *y* SEQ NOT *y*)
x must-overlap y	*x* ? ? *y*	There must be an overlap between *x* and *y* but either can start or end first	(NOT *x* AND NOT *y*) L-SEQ (*x* AND *y*) L-SEQ (NOT *x* AND NOT *y*)
x starts-with *y*	*x* ? *y*	both *x* and *y* must start together but the ordering of the ends is unspecified	(NOT *x* AND NOT *y*) SEQ (*x* AND *y*) L-SEQ (NOT *x* AND NOT *y*)
x ends-with *y*	*x* ? *y*	both *x* and *y* must end together but the ordering of the starts is unspecified	(NOT *x* AND NOT *y*) L-SEQ (*x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x starts-before *y*	*x* ? *y*	*x* must start before *y* but the ordering of the ends is unspecified	(NOT *x* AND NOT *y*) SEQ (*x* AND NOT *y*) SEQ (*x* AND *y*) L-SEQ (NOT *x* AND NOT *y*)
x ends-before *y*	*x* ? *y*	*x* must end before *y* but the ordering of the starts is unspecified	(NOT *x* AND NOT *y*) L-SEQ (*x* AND *y*) SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)
x starts-before *y* and *x* ends-before *y*	*x* ? *y*	*x* must start before *y* starts and end before *y* ends but can end before or after *y* starts	(NOT *x* AND NOT *y*) SEQ (*x* AND NOT *y*) L-SEQ (NOT *x* AND *y*) SEQ (NOT *x* AND NOT *y*)

Table 8.2: Describing partly ordered temporal relationships

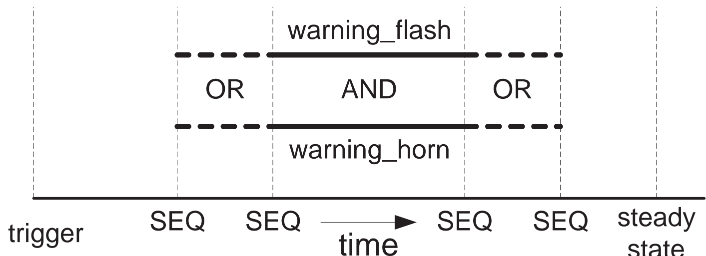



```
(warning_flash SEQ NOT warning_flash) AND
(warning_horn SEQ NOT warning_horn)
```

Figure 8.6: Two temporally unrelated effects.

The `L-SEQ` operator needs using with care where the effector components have more than two effective states, as the state of the system between the trigger and achieving the specified state are not specified, so the functional description would be satisfied if a motor jumped into running fast between being stopped and slow (for example), where slow was the required effect. Where such multiple effective states are a problem, `SEQ` and `OR` can be used to further constrain the behaviour

so that only the correct states are allowed, while leaving the order in which the intended states are entered unconstrained. This is illustrated in Figure 8.7. This




```
(NOT warning_flash AND NOT warning_horn) SEQ
     (warning_flash OR warning_horn) SEQ
     (warning_flash AND warning_horn) SEQ
 (NOT warning_flash OR NOT warning_horn) SEQ
   (NOT warning_flash AND NOT warning_horn)
```

Figure 8.7: Using OR to constrain unordered sequences

formulation means that the system cannot enter any effective states other than those specified in the functional description.

Where an intermittent or terminating effect depends on one effector with two possible states (particularly active and passive states) then the sequence and loose sequence operators are equivalent. Note, however, that despite the one change of a binary effective state in many of the temporal relationships in Table 8.1, `L-SEQ` cannot be used because the unspecified temporal intervals might include an unexpected state, such as both effects being present before the one required effect is.

While both the `SEQ` and `L-SEQ` operators will frequently be used to relate effects required for some system function, it is possible that they are used to relate subsidiary functions with their own description of purpose. In such cases, the same rule for inclusion of the consequences of failure of a function is used as for subsidiary functions combined using `AND` or `OR`, so if one of the subsidiary functions fails, that function's consequences of failure are included in the design analysis report while if more than one fail, the top level function's consequences are included. This case will generally only arise if one of these operators is used to combine purposive incomplete functions, so the sequence is launched using a single trigger.

8.3 Describing functions that depend on cyclical behaviour

The sequential operators described above are appropriate for the description of a function that depends on a sequence of effects (outputs or goal states) that terminates with the system settling into a steady state with no further triggering input. However, a system function might also depend on some sequence of behaviour that continues until some further trigger results in the behaviour terminating. A simple example is the flashing of a car's direction indicators until they are cancelled either by the driver switching off the indicators or by the self cancelling mechanism after completion of the manoeuvre.

Such functions are described by wrapping a non-terminating sequence of effects (or subsidiary functions) in a "cycle". The first state within the cycle is preceded by the `CYCLE` keyword and the state that is followed by a return to the first state is followed by the `NEW-CYCLE` marker. How this is used to describe a simple direction indicator function is illustrated in Figure 8.8. In this example, the system will

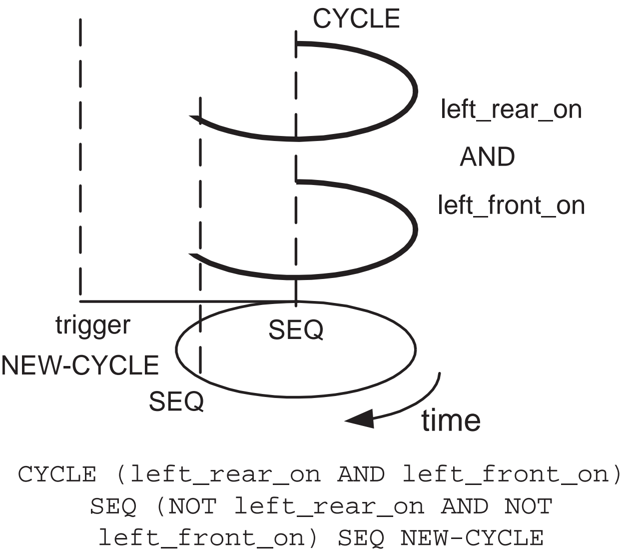


Figure 8.8: A cycle of behaviour that continues until a further trigger.

alternate between both left hand indicators being lit and unlit (simultaneously as `SEQ` is used) and this behaviour will continue until the triggering condition becomes false. In this case, the function will be triggered by the indicator switch being moved to the "indicate left" position and it will stay triggered until either the switch is moved again or the self cancelling mechanism ends the function. This might be described like this.

```
FUNCTION indicate_left
  ACHIEVES warn_of_left_turn
  BY
    indicator_switch_left
    AND NOT self_cancel_left
    TRIGGERS
      CYCLE
      SEQ (left_rear_on AND left_front_on)
      SEQ (NOT left_rear_on AND NOT left_front_on)
      NEW-CYCLE
```

in this case, the use of NOT for the effects is safe as they can be regarded as binary in nature (either the indicators are lit or not). The actual trigger conditions can be specified using IMPLEMENTS in the usual way on mapping the system model to its functional model. The implements clause for the switch might look like this.

```
indicator_switch.position = LEFT
IMPLEMENTS
indicate_left.indicator_switch_left
```

This supposes that the target system has a component called indicator_switch with the possible positions CENTRE, LEFT and RIGHT. The mapping between the functional and system models for the self cancelling feature is perhaps a little more elaborate, it could be written as follows.

```
steering_wheel.steer_left L-SEQ steering_wheel.steer_straight
IMPLEMENTS
indicate_left.self_cancel_left
```

Implementing this in the simulation might prove problematic. For modelling the self cancelling of direction indicators it is arguably sufficient to treat the steering wheel as having three positions, for turning left, turning right and straight ahead and the act of returning the wheel to straight ahead, having first turned left, will trigger the self cancelling of the direction indicators, hence the need to specify the act of turning the wheel left beforehand. If the wheel is modelled as having three discrete positions, SEQ and L-SEQ are equivalent, as it is impossible to have the wheel steering right without it first being centred in the straight ahead position. The use of these operators in specifying the triggers of functions and other temporal aspects of their description is discussed further in Section 8.4 below.

This use of CYCLE to indicate a continuing cycle of effects or subsidiary functions can be likened to a "while loop" in a programming language where the condition

upon which the loop depends is the triggering precondition of the function. This is consistent with the idea that a functional description uses an external view of the system, so the external nature of the loop's condition is appropriate. In such cases the simulation will not reach a steady state at the end of a simulation step. If the simulation is not then to continue indefinitely, the simulation engine must be capable of recognising such cases and stopping the simulation. Such requirements of the simulation are discussed in Chapter 11.4.

Another possible use of `CYCLE` is to abbreviate a long repeating sequence of effects that does terminate by associating the `CYCLE` with a count. For example, suppose a washing machine sounds a chimer three times to indicate completion of the wash cycle. This requires a somewhat repetitive functional description.

```
FUNCTION wash_complete_chime
  ACHIEVES call_attention_to_completion
  BY
    wash_complete
    TRIGGERS
      SEQ chimer_sounding
      SEQ NOT chimer_sounding
      SEQ chimer_sounding
      SEQ NOT chimer_sounding
      SEQ chimer_sounding
      SEQ NOT chimer_sounding
```

This could be shortened by using a cycle that only repeats three times instead of continually until some external trigger stops it.

```
FUNCTION wash_complete_chime
  ACHIEVES call_attention_to_completion
  BY
    wash_complete
    TRIGGERS
      CYCLE [3]
      SEQ chimer_sounding
      SEQ NOT chimer_sounding
      NEW-CYCLE
```

These formulations are identical, all this use of `CYCLE` does is shorten the description of the repeated sequence of effects. In other words, the cycle should start with the system achieving the first listed effect and the cycle should end with the system in the state associated with the effect preceding the `NEW-CYCLE` flag after the correct number of repetitions.

These sequential (and cyclical) operators introduce a temporal dimension to the functional modelling language. How this affects the nature of a trigger is discussed next while the extension of this temporal aspect to allow the language to describe untimely achievement of a function is discussed in the following chapter.

An alternative approach to modelling cyclical behaviours for interpretation of simulation might be to use an abstract state that encompasses such behaviour, so a direction indicator might have a flashing state that abstracts the cycling between on and off. This is similar to the approach in (Keuneke & Allemang, 1988). This approach is feasible provided there is no danger of a system fault (either in design or arising from component failure) that results in a behaviour that is hidden by the abstraction. For example, if we treat the indicators as either off or on (implying that when on they are flashing, the functional description will be unable to distinguish between correct system behaviour and the behaviour arising from some fault in the flasher unit that results in the indicators lighting up but not flashing.

8.4 Temporal aspects of a function's trigger

In the preceding chapters, a trigger was treated as some condition that was (and implicitly remained) true if the function was to be achieved. For example, a switch being (and remaining) closed to switch on the torch's lamp. However, the use of the sequential operators introduced in this chapter suggests that a trigger be treated as an event as much as a persistent condition, so the sequence of effects associated with a function is started in response to the event of triggering the function, which might or might not imply a persistent change of state of the component associated with the trigger. The washing machine example above is a case in point, where the three chimes are triggered by the event of completion of the wash cycle, with no specific component state being associated with this trigger. This confirmation function is actually triggered by the achievement of the washing function; such functional dependencies are discussed in Chapter 10. Implicit in this treatment of a triggering condition is the idea that the function's effects will, once triggered, run to completion. In the washing machine example, the chimes will continue, even though the user might respond to the signal (maybe by opening the door to unload the machine) before its sequence of effects is complete.

The approach taken with the functional modelling language is that once an external event makes a triggering condition true, it is considered to remain true until some other external event causes it to become false (like the self cancelling of the

direction indicators). This is consistent with the functional description viewing the system as a black box as the functional description need not take into account whether some triggering action causes a persistent change of state of any component in the system. One example to which this applies is the dip switch of some cars. In this case, the action of pulling and releasing the dip switch causes the headlamps to dip if they are on main beam or switch to main beam if dipped. This could either be accomplished by the dip switch having a toggle so that if the switch is open pulling and releasing it closes the switch and vice versa or the switch itself might simply be a "pull-to-make" passing contact switch and the pulse of current resulting from the switch beng pulled and released causes an internal component (such as an electronic control unit) to toggle between two possible states. From the external point of view, that of either the user of the system or the functional description, it does not matter which of these implementations of the expected behaviour is actually used. Notice that if the dip switch is "pull-to-make", then there is no persistent change of state associated with the switch so it is not appropriate to use the switch state as a persistent precondition for the dipped headlamps function. This is another case where a sequential operator is needed in describing a function's trigger.

This means that if a sequence of effects is to be ended by some user action, this must be explicitly included in the function description. This has already been shown, of course, in the direction indicator example used earlier, as such a cancelling condition is necessary for a cycle of effects that will otherwise never terminate, as in that case. It could, however, also be used in the case of a sequence that will terminate of its own accord if not cancelled. For example a simple electrical kitchen timer might sound a series of bleeps until cancelled or up to a certain number of such bleeps has been emitted, whichever happens sooner. This function might be described as follows.

```
FUNCTION timer_sounds
  ACHIEVES call_attention
  BY
    time_complete AND NOT cancelled
    TRIGGERS
      CYCLE [20]
      SEQ bleeper_sounding
      SEQ NOT bleeper_sounding
      NEW-CYCLE
```

The act of pressing a button on the timer will implement the cancelled trigger condition so will stop the bleeper sounding even if the expected number of bleeps have not been sounded. This raises the need to trace back through the description

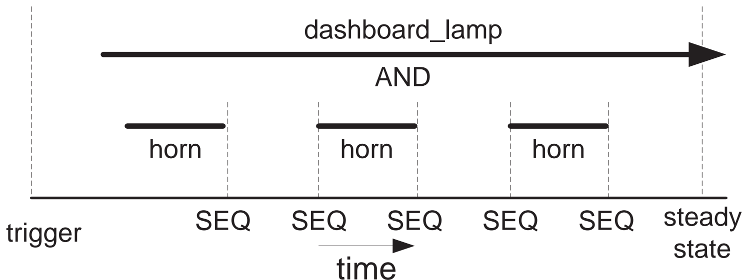
of the sequence of states arising from the simulation to establish whether or not the terminating condition becomes true, in this case whether the cancelling button is pressed. The use of these sequential operators already entails such a sequential description of the simulation for interpretation but this raises the difficulty of incorporating the later external triggering event into the simulation. This is problematic if a simulation of the behaviour is associated with a single triggering event on the system, which remains in the state it was eft in by the previous simulation. For example, in the case of the timer, a simulation will be run after starting the timer, so it should end with the timer sounding. The next simulation might be of what happens when the cancel button is pressed but then some way is needed, in the simulation tool, of specifying that this action takes place while the timer is still sounding (or not, as the case may be). In practice such a timer might well stop sounding after a certain time has elapsed since the sequence of bleeps began, if it isn't cancelled. How this is described in the Functional Interpretation Language is the subject of the next chapter.

8.5 Using the sequential operators

As the example functions used to illustrate the use of the sequential operators have been unrealistically simple, it seems worth discussing their use in the context of a more realistic case study. The seat belt warning system that has been mentioned earlier and is used as a complete case study in Chapter 11 is a useful example of a system whose functions require the use of the sequential operators. The system lights a telltale on the dashboard and sounds a horn three times if the driver or front seat passenger has not buckled their seat belt once the car has moved off. There is a fuller description of the system's intended functions in Section 11.1.1. The system's main warning function might be described like this.

```
FUNCTION unbuckled_warning
  ACHIEVES warn_seatbelt_unbuckled
  BY
    car_moving
    AND
    (driver_unbuckled OR
      (passenger_seat_occupied AND passenger_unbuckled))
    TRIGGERS
    dashboard_lamp
    AND
    (horn SEQ NOT horn SEQ horn SEQ NOT horn
    SEQ horn SEQ not horn)
```

The trigger for this function is itself quite complex and is discussed in Chapter 11 so it is not considered further here. The effects specified are as shown in Figure 8.9. Here there is no synchronisation specified between the lamp coming



dashboard_lamp AND (horn SEQ NOT horn SEQ horn
SEQ NOT horn SEQ horn SEQ NOT horn)

Figure 8.9: Using the sequence operator in the seat belt warning system.

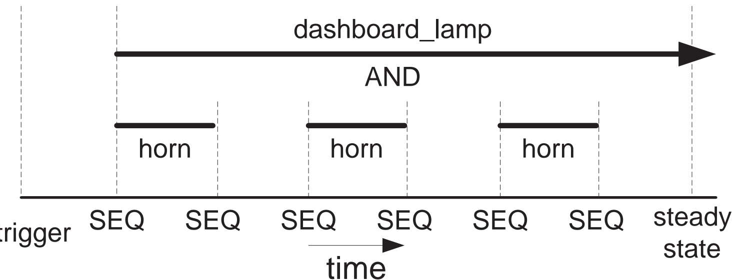
on and the starting of the horn's sequence. While such synchronisation is likely to be unnecessary, if it is to be specified the effect might be changed to

```
SEQ (dashboard_lamp
AND
(horn SEQ NOT horn SEQ horn SEQ NOT horn
SEQ horn SEQ not horn))
```

The other parts of the functional description are identical to that above so have been omitted. The lamp should come on simultaneously with the start of the first sounding of the horn. This is similar to the way the headlamps should come on to main beam simultaneously was specified above and is illustrated in Figure 8.10. Another possible improvement is to use a counted cycle to shorten the description of the horn sequence, as was discussed in the washing machine example. This will result in the following effect.

```
SEQ (dashboard_lamp
AND
CYCLE[3] SEQ horn SEQ NOT horn NEW_CYCLE)
```

There is arguably no need to add `SEQ` at the start of the sequences of horn sounding above as the horn is a single effector with a binary effect - sounding or not sounding - it is either on or off so the first sounding can be taken to immediately follow a not sounding state. However, it is needed at the start of a cycle so as to




```
SEQ (dashboard_lamp AND (horn SEQ NOT horn SEQ
  horn SEQ NOT horn SEQ horn SEQ NOT horn))
```

Figure 8.10: Synchronising the horn and lamp in the seat belt warning system.

specify the sequential relationship between the last effect in the cycle and the first effect of the next iteration. The nature of the horn's effect also means that `L-SEQ` could be used in the horn sequence but not to specify that the sequence and lamp should start simultaneously.

In all these variants of the seat belt warning functional description, purposive incomplete functions could be used in place of the individual effects so that the description captures the idea that the presence of either one of the effects mitigates the failure of the main function, as described in Section 7.1.

In practice, the horn sequence of the actual seat belt warning system is timed, so the horn sounds intermittently for a set period of time before stopping. How such cases are modelled is discussed in the following chapter.

While the discussion in this chapter has, like (Bell & Snooke, 2004), concentrated on the use of the sequential operators to describe effects of a function, which is perhaps their most common use, it is possible to use them for describing other parts of a functional description. The possibility of the effects being replaced by purposive incomplete functions has already been noted, and as these simply take the place of an effect in the top level function's description, this substitution is simple.

Where the triggering of a function depends on two actions taken in a certain order, the sequential operators can be used to specify the intended order. The steering left and returning to straight ahead to implement the self cancelling of the direction indicators is a case in point as is the pulling and releasing of the dip switch in the dipped headlamps example discussed in the previous section. Either `SEQ` or `L-SEQ` might be used as appropriate. One interesting example where `L-SEQ` is appropriate is the dip switch example as the dip switch will typically be on the stalk attached to the steering column housing that also switches the indicators.

In such cases it will presumably be the case that the operation of dipping the headlamps should still be triggered even though the indicators are switched on (or off) between the pulling and releasing of the dip switch.

It is also possible to use these operators to specify a sequence of complete subsidiary functions (or operational incomplete functions) that must be achieved (in the specified order) to achieve the main function. A possible example of this use of these operators might be the signalling example discussed earlier, in Section 6.2.1. The functional description of the main function used there was as follows.

```
FUNCTION accept_train
  ACHIEVES guide_to_platform
  BY
    FUNCTION set_points
    AND
    FUNCTION set_signals
```

This, of course, merely specifies that the points and signals should both be set and does not specify in which order these operation be carried out. It might be considered necessary for the functional description to specify that the points be set first in which case, the two subsidiary functions will have to follow each other in the right order, like this.

```
FUNCTION accept_train
  ACHIEVES guide_to_platform
  BY
    FUNCTION set_points
    SEQ
    FUNCTION set_signals
```

In practice, this is unlikely to be necessary as a railway signalling system will incorporate safeguards to prevent the signals being set unless the points are already set, but this does serve as an illustration. In this case SEQ was used to specify that no other functions should be triggered between the two specified. Again here it is likely that triggering some other function will be impossible, or will render completion of the sequence impossible because of the interlocking of the signalling system. For example if some other route is set between setting the points and signals, it is likely that the signals can no longer be set in case there is a conflict between the newly set route and the original route set as part of the present function.

It will be appreciated that these sequences of events are likely to be constrained by the timing of individual effects (for example, each flash of a direction indicator

must last a certain period). How these timing constraints are represented in the functional modelling language forms the subject of the following chapter.

Chapter 9

Timing and function

The introduction of representation of functions that depend on sequential effects, leads naturally on to the idea that a functional description might be required to specify the duration of each element in such a sequence. An example might be so that the duration of each flash of a sequence of flashes can be specified. Another reason for specifying the timing of achievement of a function is so that the functional description can be used to highlight cases where a function is achieved correctly (that is, in response to its trigger) but in an untimely manner. This will typically be because the effect is delayed, so the function is achieved late, but might be because an expected delay is absent (or reduced) so the function is achieved early.

The increasing use of networks for the transmission of control signals and data is a possible cause of change to the timing of the achievement of the expected effect of some function. This is, of course, especially likely if the network is of non-deterministic message latency, such as the collision detection and resolution protocol used in the CANbus (Bosch, 1991) common in the automotive sector. In such carrier sense multiple access collision detection (CSMA/CD) networks, any node can transmit a signal at any time so if two nodes attempt to transmit simultaneously the signals collide. Unlike in the CSMA/CD protocol used in Ethernet, in CANbus and similar protocols the transmitters are given a priority rating and so only the lower priority message is lost and must be re-transmitted later. This does, of course, result in a delay to that message and this might in turn lead to a delay in response to a control input. Therefore heavy use of the network could conceivably lead to undue delays in response to relatively low priority operations. For example, where the driver's control operations for a car's lighting are transmitted between electronic control units by such a protocol, some delay in, say, the headlamps dipping might be encountered.

Where a function's effect depends on a sequence of outputs (or goal states) then it is also possible that the timing of the members of this sequence is significant. A simple example is the timing of a car's direction indicators. A fault in the flasher unit might result in the flashing being unexpectedly rapid, which failure is of some significance if only because there are legal restrictions on the timing of these flashes.

These two cases highlight the need for a functional description to be able to specify any requirements in timing so that the design analysis tool can indicate that these are not being met, if some function is achieved late, for example. These specifications of requirements in timing of achievement of a function are referred to as "temporal constraints" (on correct achievement of the function). This chapter discusses how they are specified in the functional modelling language and then some questions that arise are discussed in the light of the chosen approach. Alternative approaches are considered in that discussion.

9.1 Representation of temporal constraints

As stated above, rather than discussing alternative approaches to specifying temporal constraints, the chosen approach will be described next, allowing it to be used as a starting point for this discussion. This description will be centred on the specification of timing of effects of a function relative to the event that triggers the function, as this is the most common use. This does, of course, imply the presence of a triggering event which can be used as the basis for timing the achievement of a function's effects. This is similar to the idea that a trigger starts the first time slot in a simulation step in the use of the sequential operators as discussed in the previous chapter. So, for example, rather than a switch being closed triggering the function, for the present purposes, the action of closing the switch (which can be regarded as momentary) is what triggers the function.

Where a function's timing is significant there are naturally two ways that it can deviate from the specified timing. It can either be too late or, more rarely, too early. The most common case of a function's effect occurring too early is when it results in a terminating effect ending too soon, such as a light's flashes being made too short. To distinguish between these cases, a temporal constraint can have either one or both of these two keywords.

- `AFTER`, is used to specify a minimum delay before the effect should be achieved.

- BEFORE to specify a deadline by which the effect should be achieved.

One or other of these constraints, or both, are added to the functional description immediately following the element to which they apply. They are separated from the functional description itself by being placed in square brackets. Each temporal constraint is followed by the specified time itself. For the purpose of this description numerical values of time are used but possible approaches to representation of time are discussed in Section 9.3 below. While these constraints are likely to be used with the sequential operators described in the last chapter, adding expected durations to the time slots into which these operators divide the intended behaviour, they can be used together with other operators, or, of course where no logical operator is required. A simple example might be the headlamp dipping function of a car where CANbus is used to carry control messages in the lighting system, introducing a the possibility of delay in response to the driver's action of moving the dip switch. This might be described like this.

```
FUNCTION dipped_beam
  ACHIEVES light_road_without_dazzling
  BY
    lamp_switch_heads AND dip_switch_dipped
    TRIGGERS
    left_dipped
    AND [BEFORE 250ms]
    right_dipped
```

This specifies that for the function to be achieved in a timely manner, the post-condition `left_dipped AND right_dipped` must resolve to true within 250 milliseconds of the precondition becoming true. This is illustrated in Figure 9.1. The

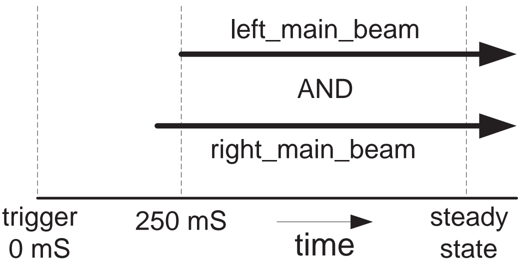


Figure 9.1: A simple temporal constraint on achievement of a function.

time constraint refers to the achievement of the required effects, not to the end of the simulation, which will typically follow the achievement of the effects more immediately than in the figure. If the post-condition does not become true until after the time limit, the function is achieved late. Therefore the functional description

allows the design analysis to distinguish between cases where the function fails (the post-condition does not become true) and where the function is achieved, but late. In this case, there is no need to specify a time that should elapse before the effect is achieved, so `AFTER` is not included in the temporal constraint. Note that the position of the temporal constraint is significant. For the constraint to be met, the condition specified by `AND` must be true, so both headlamps must dip in time. As was discussed in the previous chapter, this description does not specify that the headlamps actually dip simultaneously merely that they have both done so within the time specified. The sequence operator `SEQ` can be used if it is required that the headlamps do dip simultaneously, as discussed in Section 8.1. If the temporal constraint is placed after one of the effect labels, it would apply only to that label. This is unlikely to be appropriate in the headlamp example, but a warning system that sounds a horn and light a lamp might have the effect described like this.

```
light_telltale
AND
sound_horn [BEFORE 250ms]
```

In this case, the horn must sound within 250 milliseconds but the lamp treated as not being subject to any temporal constraint, perhaps because it is the horn that first draws attention to the subject of the warning. Note that to save space, the example functional descriptions in the rest of this section will only include the effect as that is where the temporal constraint will generally be added. However, there are cases where a trigger might consist of a timed series of operations (such as switch being pulled and released within a certain time. This will be discussed further.

The most common use for the `AFTER` constraint is to specify a minimum duration for a transient effect (such as a flash). Suppose, for example, that the warning system above was to sound its horn three times and it was necessary to specify the duration of each sounding. This might be done as follows.

```
light_telltale
AND
SEQ sound_horn [BEFORE 250ms]
SEQ NOT sound_horn [AFTER 800ms BEFORE 1200ms]
SEQ sound_horn [AFTER 500ms BEFORE 700ms]
SEQ NOT sound_horn [AFTER 800ms BEFORE 1200ms]
SEQ sound_horn [AFTER 500ms BEFORE 700ms]
SEQ NOT sound_horn [AFTER 800ms BEFORE 1200ms]
```

As the sequential operators are unary, placing the temporal constraint after the operator means the same as placing it after the effect label or expression to which the operator applies. The temporal constraints constrain the timing of the ends of each interval in the temporal description, as illustrated in Figure 9.2. In this case,

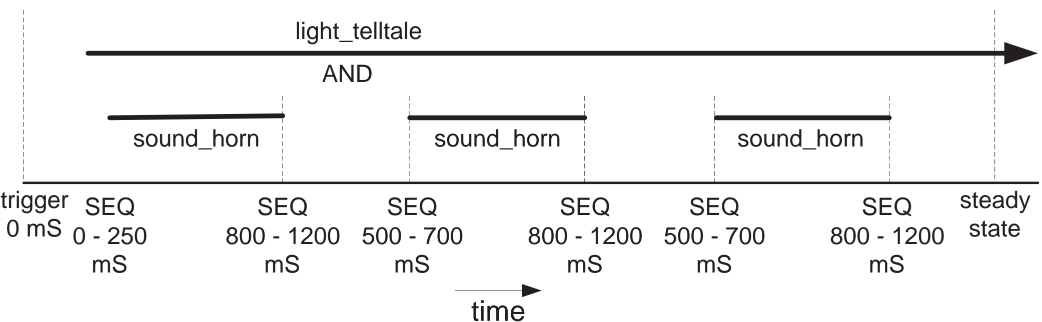


Figure 9.2: Adding temporal constraints to a sequence of effects.

the horn should first sound within 250 milliseconds of the trigger, and should stop sounding after at least 800 milliseconds but before 1200 milliseconds and so on. The temporal constraints of some event in the sequence specify the duration of the preceding interval. In general, a range of times for the events will be specified (as here), so as to avoid the danger of insignificant changes of timing being recorded on the resulting design analysis report. It will be seen that each time constraint is timed from the previous one. This is necessary because if there is any tolerance in the required timings, as in this example, then these tolerances will accumulate during the sequence, as illustrated in Figure 9.3, which contrasts the tolerances of timing of the horn on and off intervals using the two possible ways of specifying the timings. In the top line of the figure, the timings are specified relative to

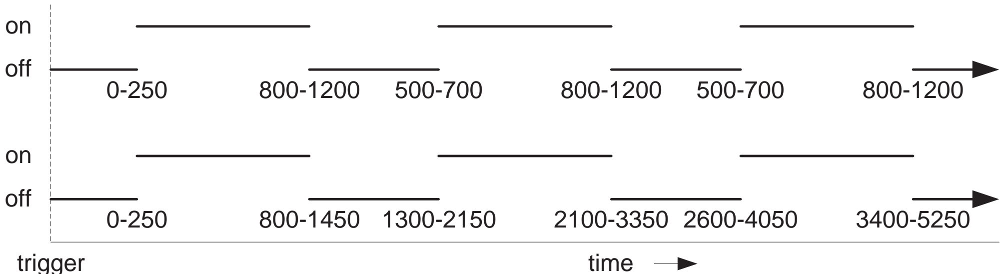


Figure 9.3: Accumulation of tolerance if timing is taken from the initial trigger.

the previous event (starting with the trigger) while in the lower version, the same range of timings is used, but all relative to the trigger. It will be seen that to preserve the degree of tolerance of variation of duration of each period the overall tolerance gets so large as to allow any individual period to have a duration of zero.

This specification of timing is simple where the strict sequence operator (SEQ) is used, of course, because the transition between the preceding and succeeding intervals is treated as instantaneous so the temporal constraint is unambiguous. However, the loose sequence operator (`L-SEQ`) has to be used with care where time constraints are to be used. In this case, the time constraint naturally applies to when the operator becomes true, so to the end of the undefined transitional interval associated with `L-SEQ`, as illustrated in Figure 9.4. This has the drawback that when successive effects are joined using `L-SEQ`, the timing of the later effect cannot be used to limit the duration of the preceding effect, as the timing of the start of the undefined interval implicit in the use of `L-SEQ` is not specified. A possible example might be an automatic machine tool where the machining should not start until the workpiece has reached the correct operating speed after starting the motor and the correct operating speed has to be maintained for a sufficient period to allow the machining to be completed before stopping the rotation so the workpiece can be removed from the machine. In this case, `SEQ` cannot be used to join the successive goal states (stopped and running, where running is defined as running at the correct operating speed) as there will be a period during which the piece is accelerating and another when it is slowing down. If `SEQ` is used (implying the acceleration and deceleration are instantaneous) than the result of the simulation will be "out of step" with a simulation that includes the correct sequence of states. It will also lose either the distinction between running at the correct operating speed and some other speed, or running at this (slower) speed and being stopped. This sequential effect might (naïvely) be described using `L-SEQ`, as shown below, with the time constraints that the workpiece should reach the operating speed in 3 seconds and run for at least 8 seconds.

```
stopped
L-SEQ running [BEFORE 3s]
L-SEQ stopped [AFTER 8s]
```

This is illustrated in Figure 9.4 where it will be seen that as the time constraints apply to when the operator becomes true, there is no specification of when the deceleration should start. The duration of the `running` state is undefined, the same functional description would be fulfilled by the case in Figure 9.5 which differs in having the deceleration taking longer at the expense of the duration of the period the machine is running at the correct operating speed. Indeed, in this case the functional description would be fulfilled be the workpiece momentarily reaching the operating speed and immediately starting to decelerate.

There are various ways of avoiding this problem. The simplest is arguably to add additional effective states (accelerating and slowing) so as to allow SEQ to be

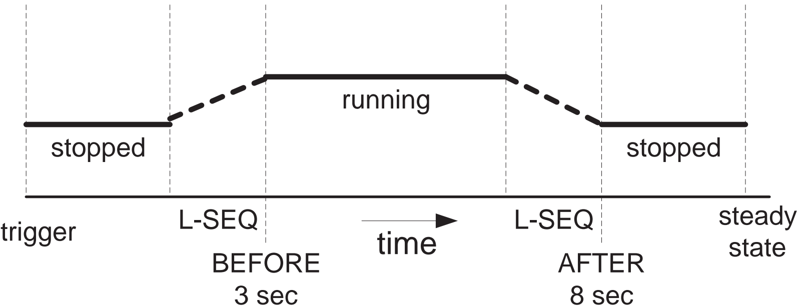


Figure 9.4: Using L-SEQ to specify timing of a sequence.

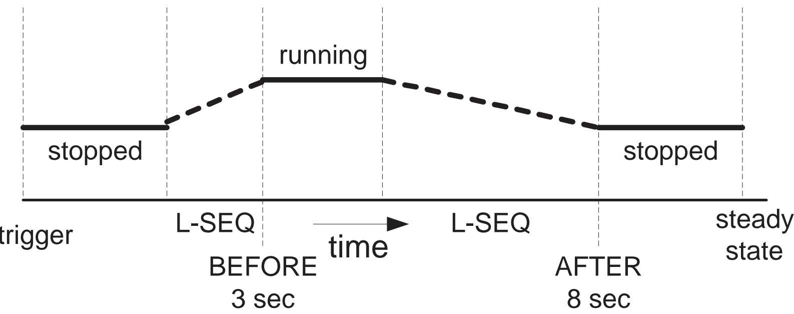


Figure 9.5: L-SEQ does not specify duration of preceding effect.

used, as now the transitions between the intended goal states can be treated as instantaneous, though at the expense of complicating both the functional description and the mapping between it and the system model. The resulting description of the function's effect might look like this.

```
stopped
SEQ accelerating
SEQ running [BEFORE 3s]
SEQ slowing [AFTER 8s]
SEQ stopped
```

In both this description and that above, the timing of the sequence starts with the trigger becoming true. As temporal constraints are taken from successive timed steps, the amount of time spent accelerating is undefined, provided the operating speed is reached within 3 seconds, and the time taken for the workpiece to come to a stop is also undefined. Temporal constraints could be added to either or both of these events.

In some cases, `NOT` could be used to avoid the definition of additional states, as shown below.

```
NOT running
SEQ running [BEFORE 3s]
SEQ NOT running [AFTER 8s]
```

This loses the distinction between accelerating and slowing and stopped, significant because it is only once the workpiece is stopped that it can be removed from the tool. A possible, if slightly cumbersome, solution might be to use `NOT running` in the place of accelerating and slowing, though `stopped SEQ NOT running` is rather a meaningless distinction, since if stopped is true, so is `NOT running`. However, it is not the case that `stopped` is necessarily true when `NOT running` is true, so the distinction between those states is not itself meaningless.

The language could be extended to avoid this difficulty with the use of `L-SEQ` and temporal constraints. Two possibilities are to add a duration to allow the temporal constraint to be associated with an interval rather than a transition event. This is rejected as in general, such durations can be defined using a combination of `BEFORE` and `AFTER`. Using durations of intervals in the temporal model to specify temporal constraints is discussed in Section 9.2. An alternative would be to allow an `L-SEQ` operator to be associated with two (pairs of) temporal constraints, one of which specified when the undefined interval should start and one that defines when it should finish. Either or both could be specified for a given instance of `L-SEQ`, so for the example used above, the description of the effect might be as follows.

```
stopped
L-SEQ running [END BEFORE 3s]
L-SEQ stopped [START AFTER 8s]
```

Here `START` is used to indicate that the undefined period starts consistently with the temporal constraint and `END` that that interval should end consistently with the constraint. In this case, then, the workpiece reaches the operating speed before 3 seconds have elapsed since the trigger became true and the (implicit) slowing of the workpiece will not start for another 8 seconds. They could, of course, be used together to restrict the duration of the undefined period associated with `L-SEQ`. These additions to the language have not been adopted as they add no expressiveness over and above the careful use of `SEQ`, as outlined above.

The warning system mentioned earlier is a case where a cycle might be used to avoid a repetitive description of the horn sounding. This raises the difficulty that the first sounding of the horn will typically have a different temporal constraint from the later ones, that take place within the cycle. The approach taken is to associate the first timing constraint with the start of the cycle, as shown here.

```
light_telltale
AND
CYCLE[3] [BEFORE 250ms]
SEQ sound_horn [AFTER 500ms BEFORE 700ms]
SEQ NOT sound_horn [AFTER 800ms BEFORE 1200ms]
NEW_CYCLE
```

This is perhaps not completely clear. The rule is that the cycle begins when the first effect (or expression) listed within the cycle is true (so in this case, the horn starting to sound) which in this case should happen within 250 milliseconds of the trigger. The timing associated with that first effect are for its later occurrences within the cycle. In other words the cycle's temporal constraint overrides the first effect's temporal constraint for the first occurrence. Naturally, this approach will work either for a counted cycle as here or to specify a timing for a cycle that is to continue until some future triggering event causes it to stop.

Another possible case that involves timing of a cycle is where the cycle should continue for a specified period of time before stopping. For example, a horn might sound intermittently for a period of (say) between ten and fifteen seconds after which it stays silent. This can be described by specifying the state that follows the cycle, like this.

```
light_telltale
AND
CYCLE [BEFORE 250ms]
SEQ sound_horn [AFTER 500ms BEFORE 700ms]
SEQ NOT sound_horn [AFTER 800ms BEFORE 1200ms]
NEW_CYCLE
SEQ NOT sound_horn [AFTER 10s BEFORE 15s]
```

This specifies that the state following the cycle should be achieved between ten and fifteen seconds following the start of the cycle. It s unaffected by the timing constraints within the cycle. This is perhaps not dissimilar to the scoping of variable within an iteration in a program. There is, of course, a possible difficulty here as the cycle might correctly stop after ten seconds with the horn already silent so the cycle appears to have ended early, despite the absence of any fault. The simplest approach to this problem is simply for the resulting design analysis report to show the apparent early end to the cycle and as it arises from no malfunctions for it to include timing tolerances with the cycle as the cause of the apparent fault. In the description of `CYCLE` in the previous chapter, the functional description is taken to remain in the state immediately before the `NEW-CYCLE` operator once the cycle finishes. Of course, where a cycle's end is triggered by an external timer,

any of the effects listed within the cycle might happen to be active. Therefore the effect that is expected after the time has passed is specified in the functional description, as in the example above. This specification of the terminating effect will generally not be needed when the cycle is dependent on the truth of some triggering condition, such a switch position.

It is appreciated that the construction used for specifying temporal constraints is not easily read, as the timings associated with an expression that includes some effect are those for the duration of the preceding effect, but this could be hidden from the user and the meaning, though perhaps unclear, is unambiguous. An alternative approach would be to specify a duration for each effect, so associating timings with intervals rather than transitions, but this introduces complications of its own. This alternative approach is discussed in Section 9.2, that discussion includes reasons for preferring the chosen approach, as described here.

While this discussion of temporal modelling has been concerned with the timing of a function's effects, it is entirely possible that a function might be triggered by a temporally constrained sequence of events. A simple example is where shutting down a personal computer requires the power button to be pressed for a certain length of time to avoid the danger of an accidental knock to the button resulting in an unintended shutdown. This can be described like this.

```
FUNCTION shut_down
  ACHIEVES safely_power_down
  BY
    press_power_button SEQ release_power_button [AFTER 5S]
    TRIGGERS
    close_down_computer
```

It is appreciated that the effect will typically be a good deal more complicated than shown here. This raises the difficulty of how the simulation manages these temporal triggers. Typically, the simulation runs in steps, each triggered by a single triggering action (such as pressing the power button) but this is not appropriate here. Short of having a real time simulation there is no ideal solution to this problem. One possible answer is to work around the problem by having the two actions combined as one trigger, but this does not allow the system to model what happens when the button is not pressed for long enough. Of course, even though the interpretation of simulation might be unable to take full advantage of this extra expressiveness for this reason, it is quite possible that such temporal triggers are included if the language were used for refinement of a functional specification in design of a system.

9.1.1 Consequences of untimely achievement of function

The advantage of the temporal constraints described here is that the functional model can distinguish between failure of a function and its untimely achievement. One consequence of this is that the resulting design analysis report will need to distinguish between the consequences of untimely achievement of a function and its failure. These will typically differ, with the untimely consequences being (in general) less severe as actual failure.

The suggested approach to this is to allow the mode builder to specify a separate set of consequences for late or early achievement of a function if these are felt necessary. If these are not added to the model, the design analysis report can still include details of the temporal failure (such as the function being achieved late), but without any consequences. Where no specific consequences are added, the value for severity can be treated as having the value 1 (the lowest possible, so it will result in the temporal failure having a low risk priority number. In many cases this would seem to be appropriate, so there might be no need to demand the extra work of the model builder, in adding the additional consequence. However, the possibility of adding the additional consequences exists where the consequences of temporal failure of a function are felt to be sufficiently significant. This is illustrated in Section 9.4, below.

9.2 Temporal constraints using duration of intervals

The approach adopted, as described above, associates temporal constraints with the transitions between system states. Therefore, the temporal constraint for the simple headlamp example specifies that the transition between the headlamps not being dipped and being dipped must be completed before 250 milliseconds have passed since the trigger condition became true. An alternative approach would, of course, be to specify the temporal constraints in terms of the time the system should remain in a certain state, so specifying them in terms of the intervals rather than the transitions of the temporal model. This might be done either instead of or as well as the chosen approach.

As observed above, the use of intervals addresses the difficulty of reading the functional description in the rather un-intuitive layout described above. This difficulty is especially apparent in reading the examples that include a cycle of successive effects. For example, the simple timed cycle might be described as follows.

```
light_telltale
AND
CYCLE[3]
SEQ sound_horn [>= 800ms <= 1200ms]
SEQ NOT sound_horn [>= 500ms <= 700ms]
NEW_CYCLE
```

Where 'greater than or equal' and 'less than or equal' are used to specify the minimum and maximum tolerable durations respectively. There is a considerable problem, however, with using this approach instead of the preferred association of temporal constraints with transitions. This is the difficulty of specifying the timing of the undefined interval between the trigger and the first effect. In the example here, there is an interval between whatever action triggers the warning function and the lighting of the telltale and the start of the horn sounding cycle. There is no label for this interval with which the temporal constraint can be associated and, worse, there are actually two concurrent intervals, only one of which (in this case) has an associated temporal constraint. As the state of the effects of the system are derived from the previous function (before the trigger event) the label for this initial interval cannot be unambiguously defined. Consider the dipped headlamps example from earlier in the chapter.

```
FUNCTION dipped_beam
  ACHIEVES light_road_without_dazzling
  BY
    lamp_switch_heads AND dip_switch_dipped
    TRIGGERS
    left_dipped
    AND [BEFORE 250ms]
    right_dipped
```

Here, the trigger depends on two switches and in general one of the two conditions will be true before the trigger itself becomes true. Either the dip switch will be in the dipped position with the headlamp switch set for sidelights only or the headlamps switch is on and the dip switch set to main beam. Therefore the state of the system at the start of the functional description is unknown. Therefore there is no easy way of attaching a label to this initial interval. In this case the headlamps should be either off or on main beam, which could be combined (using `OR`) to specify the initial state, but this will not always be the case. At least in this case the initial interval has a clearly defined end. In many cases this might well not be so. Consider the simple warning system, where a temporal constraint was associated with the sounding of a horn, but not with the lighting of the telltale lamp. To describe this in terms of intervals requires two (concurrent) initial intervals, labelled (presumably) with the assumed state of the respective

effects to allow the temporal constraint associated with the horn starting to sound to be specified. This might be described like this.

```
some_malfunction
TRIGGERS
NOT light_telltale SEQ light_telltale
AND
NOT sound_horn [< 250ms] SEQ sound_horn
```

There seem to be several objections to this. One obvious one is the additional complexity of the description compared with the equivalent description specified in terms of transitions. Another is the inclusion of effects that are not properly associated with the trigger. The trigger here does not trigger the lamp being off and coming on, as the description appears to state, rather it is assumed that the lamp is already off. Of course it might not be, either because of some malfunction such as a wire shorting to ground or because some other function that is already achieved required the same lamp to be lit.

In addition to this difficulty with the use of intervals, the approach does not solve the difficulty with the use of loose sequence (`L-SEQ`). `L-SEQ` implies an undefined interval, in the functional description and this interval does, of course, remain undefined whether intervals or transitions are used as the basis for specifying temporal constraints. The machine tool example used above might be described like this using intervals.

```
stopped [< 3s]
L-SEQ running [> 8s]
L-SEQ stopped
```

The same objection applies here. An interval will be regarded as starting once the condition becomes true and continues until the condition is no longer true so the machine is regarded as stopped until `L-SEQ` running is true (which must happen within three seconds) and this interval continues until `L-SEQ` stopped becomes true, so there is still no specification of the duration of acceleration and deceleration, so the period the machine is running at the operating speed is undefined, just as is the case when transitions are used. The difficulty here is that there needs to be a mapping between the operating speed (a numerical value) and the state (a Boolean value). This can be managed by defining the acceleration and deceleration as distinct states as described earlier.

The use of intervals rather than transitions still implies a model of time consisting of intervals separated by (instantaneous) transitions. The timing of the start and

end of the interval being regarded as instantaneous. This is arguably an inevitable result of treating an effect as a Boolean (on or off) property of the system. Suppose an effect was a sinusoidal wave, such as perhaps a varying frequency in an audible warning). The functional description might simplify this by treating the sound (in this case) as being of high or low frequency. There will, of course, then be an instantaneous transition between the high and low frequencies (or frequency ranges) at some point on the upward and downward slopes of the curve.

It will be seen from the foregoing that while the use of intervals as the basis for temporal constraints does have the advantage of making the functional description clearer in the case of functions that use timed sequences of effects, there are considerable disadvantages of that approach, so the use of transitions is preferable. As the two approaches depend on a similar model of time (of intervals divided by instantaneous transitions) they could be mixed, by being offered as alternatives. This is not done merely to simplify the language.

9.3 Possible representations of timings

While the examples used in this chapter have all included a numerical representation of time, it seems natural to allow for the possibility of a qualitative model of time, if only for consistency with the possible use of the Functional Interpretation Language in concert with a qualitative simulation. This section discusses approaches to this, following a brief discussion of some apparent difficulties with temporal constraints.

Where time is specified numerically, as above, this leads to the introduction of a hard deadline which, if met is fine, but if missed results in the temporal failure of the function. In the headlamps example if the lamps dip in 250 milliseconds, this is fine but if they are found to take 251, the function is deemed to be achieved late. There is no way of avoiding this problem, it is at least in part a consequence of blending a numerical model with a qualitative one, where a function is either achieved or not. The argument that "if 250 milliseconds is OK why not 251?" breaks down, of course because it can just as well be applied to 251 and 252, and so on, so some arbitrary end point is required. Using a qualitative model of time provides a partial answer to this question, as will be discussed later.

The use of a range of acceptable values for a temporal constraint does mitigate this, but does not eliminate it, of course. In the horn sequence above, the horn can stay on for any time between 800 and 1200 milliseconds, so some deviation from the supposed intended duration of one second is tolerated, but still 1200

milliseconds is acceptable and 1201 is not.

An alternative approach to specifying a range of times might be to start with the intended time and allow a variation, so instead of specifying the end of the horn sounding as `AFTER 800ms BEFORE 1200ms` it might be specified as “1000ms +- 200ms”. Naturally, different times could be specified for each direction of variation or they could possibly be shown as a percentage variation. These are not adopted as they offer no real advantage over the approach described earlier. Note that they do not address the hard deadline problem. “1 second + 10 percent”, say, still means that 1100 milliseconds is acceptable while 1101 is not, so all these variations do is disguise the hardness of the deadline.

As was noted, a qualitative model of time might be a way of mitigating this difficulty. Approaches to qualitative modelling of time are discussed next.

9.3.1 Qualitative modelling of time

There are two possible qualitative models of time that might be considered. One is to specify times by order, so some effect must be achieved before some other event in the simulation. This is not appropriate in most cases, including all the examples discussed in this chapter. One case where it might be is a car engine management system, where the system has to decide on the timing of the spark for ignition before a certain point in the cylinder’s crank cycle. Is is possible even here that this will not be regarded as a functional description because the behaviour of the engine management system might simply be incorporated into the simulation of the engine. If this is not done, then the timing of the cycle must be represented as some sort of deadline. It is therefore suggested that this approach to modelling time is not really appropriate for functional description, where function is regarded as modelling an external view of some system.

The other approach to qualitative modelling of time is to use an order of magnitude approach, following (Raiman, 1991). Here changes of timing within one order of magnitude are allowed, but not changes of timing that cross into another order of magnitude. This is consistent with Raiman’s approach, where an order of magnitude is regarded as negligible to the next bigger order of magnitude. For example, a function might be specified as being achieved if the effect is true within milliseconds, but not if it takes seconds. This is used with qualitative behavioural modelling in the design analysis tool developed at Aberystwyth. Here a relay (for example) will have a delay of milliseconds between the coil becoming active and the switch closing. Therefore some system function that depends on the relay will

be achieved within milliseconds. If some fault causes the system function not to be achieved until seconds have passed the function is achieved late.

One difficulty with this approach for specifying temporal failures of functions is the difference between likely order of magnitude. A function that should be achieved in, say, 50 milliseconds might take 500 and still be within the order of magnitude despite the tenfold increase in the delay. This approach arguably depends on the order of magnitude behavioural modelling. If it was used to draw qualitative distinctions between different periods where the simulation itself uses a numerical model of time, the question of where the borders of the qualitative periods are placed arises. Is a period of 500 milliseconds counted as milliseconds or as half a second? There must then be such boundary times. Maybe 499 milliseconds is treated as belonging to the millisecond interval while 500 is treated as half a second in the seconds interval. It will be seen that this reintroduces the strict deadline problem discussed earlier, but there are fewer of these deadlines. However, they are not specified by the model builder on a case by case basis, the model relies on the assumption that any delay within a specific order of magnitude time slot is acceptable and any that crosses the boundary between adjacent time slot is not.

It would be possible to introduce a bigger set smaller intervals, perhaps, such as milliseconds, tens of milliseconds and so on. This increases the sensitivity of the temporal functional description at the expense of introducing more boundary times.

9.4 Timing constraints in use

While the discussions above have been illustrated using simple examples and more complex examples of the use of the functional language are discussed in Chapter 11, it seems worth providing a simple illustrative example of how the temporal functional modelling might be incorporated into design analysis.

If we reuse the simple headlamp dipped example, where the vehicle's lighting system makes use of a CANbus (or similar network protocol) to pass control messages across the lighting system. This raises the possibility that a delay to a message, because of its being retransmitted after collisions, results in a delay in achieving a function's effects. The functional description has been shown before, but as a reminder it is reproduced here.

```
FUNCTION dipped_beam
```

```
ACHIEVES light_road_without_dazzling
BY
  lamp_switch_heads AND dip_switch_dipped
  TRIGGERS
  left_dipped
  AND [BEFORE 250ms]
  right_dipped
```

So once the trigger condition becomes true (either by switching on the headlamps or by dipping them) both headlamps should be dipped within 250 milliseconds. There are, of course, different possible consequences depending on whether the headlamps were off or on full beam before. This is mitigated by the fact that a headlamp remaining on full beam will be treated as an unexpected effect. An extract from the resulting design analysis report might look like that in Table 9.1. In the table, it will be seen that the consequences of a failure (excess traffic on

Failure effect	**Cause**	**Consequence**	**Sev**	**Det**
When dipswitch_dipped, function headlamps_dipped failed.	wire A open circuit	Road ahead unlit	7	2
When dipswitch_dipped, function headlamps_dipped achieved late	excess traffic on CAN	Oncoming drivers momentarily dazzled	3	4

Table 9.1: Part of a design analysis showing late achievement of a function.

the network) that result in late achievement of a function are distinguished from those of one that result in its failure. In this case, the temporal failure assumes that the headlamps were on full beam, and properly the report might include this. It has been omitted for simplicity.

As was suggested in Section 9.1.1 above, it might be felt that the consequences of temporal failure of a function are not so severe as to require the inclusion of an additional set of failure consequences, so they might be omitted. If this were done in this case, the corresponding extract from the resulting design analysis report might look like that in Table 9.2. Here default values of 1 (the lowest possible) are used for the indicators of severity and detection. This results in the failure having a low risk priority number (RPN), of course.

Incidentally this example raises a difficulty with modelling systems that depend on a network as the failure that results in the late achievement of the function might well be outside the system, such as some other node on the network transmitting more often than intended. One possible approach to this in simulating

Failure effect	Cause	Consequence	Sev	Det
When dipswitch_dipped, function headlamps_dipped failed.	wire A open circuit	Road ahead unlit	7	2
When dipswitch_dipped, function headlamps_dipped achieved late	excess traffic on CAN		1	1

Table 9.2: Part of a design analysis showing late achievement of a function with no specified consequences.

the system is, of course, to treat the network as a component, one of whose failure mode behaviours is to delay messages. A more sophisticated approach might be to combine the simulator with a tool capable of modelling the network so that possible heavy loading resulting in message collisions can be modelled. To do this realistically, however, implies a model that simulates all the systems associated with the network, so that loading are correctly simulated. It also seems to imply running the simulation in real time (or a scale model of real time).

The inclusion of temporal constraints discussed in this chapter allows functions that have to be achieved in a certain time to be described and their temporal failures (such as being achieved late) to be distinguished from failure and also increases the precision of description of functions that depend on sequential effects, as described in the previous chapter. The remaining class of function that the language should be able to describe is the class of functions that depend on the state of some other system function. These functions and how they are described form the subject of the next chapter.

Chapter 10

Dependencies between system functions

This chapter discusses classes of function that depend on the state of some other system function and how the Functional Interpretation Language can be used to describe such functions. Previous chapters have dealt with operators that allow triggers and effects (or subsidiary functions) to be combined so as to describe a single top level function with a clearly defined purpose. In other words triggers, effects and / or subsidiary functions are combined in a single functional hierarchy (that might have a temporal dimension described using the operators and temporal constraints already discussed) so they are all concerned with the fulfilment of some purpose that is associated with the top of the functional hierarchy. However, there are many cases where a system includes functions (or effects) that achieve some distinct purpose from some system function but are related in some way, and which might well be triggered by the achievement or otherwise of that other function. A warning lamp that lights when some expected system output is not achieved is a simple case in point. Here, the warning lamp does not contribute to fulfilling the purpose of the main function (that which the combination of trigger and effect was expected to fulfil) but rather it has its own purpose, being to draw attention to the failure of the main function. The idea that some effect might be associated with some other function's effect but with a distinct purpose is simply illustrated by a car's blue dashboard lamp that shows the headlamps are on full beam. Clearly this lamp contributes nothing to the purpose of lighting the road ahead (it is too feeble and in the wrong place!) but there is also an association between that lamp and the headlamps themselves. In practice, of course, this is simply that it shares a trigger with the headlamps main beam's function and, indeed, it might well be regarded as being part of a separate system; the dashboard electrics as opposed to the exterior lighting system. However, if such a warning system is

more sophisticated, it cannot be tested independently of the main system as the working of the main system will affect the working of the associated system. For example, if the dashboard light only lit if the headlamps were actually lit on main beam rather than that effect being triggered. In this chapter the term dependent function is used to refer to a function that relates to and depends on the state of some main function.

The thesis will discuss four categories of "dependent function" that are to be considered in addition to the function that fulfils the main purpose of the system.

This function (referred to here as the "prime function") will, typically, be the function on which the dependent function depends. These four categories are:-

- Warning or telltale functions whose purpose is to increase the detectability of achievement or otherwise of the core function, such as the warning lamp mentioned above.
- Fault tolerant functions that allow the system to continue to (at least partially) fulfil its purpose despite the failure of some prime function. A "limp home" function, such as a car's brakes reverting to conventional operation if the sensors for anti-lock braking fail is a case in point.
- Interlocking functions that enforce correct achievement of one function's effects to ensure that some other function does not have unintended consequences. A railway signalling system is an example, where it is a function of the system to prevent setting the signals for a route unless the points are already set for that route.
- Recharging functions that return the system to a state ready for a repeat of the prime function, such as the refilling of the cistern after the flushing of a W.C.

This chapter will discuss each of these classes of function in turn, considering their relationships with the core function and how the functional modelling language can describe them, as well as any other problems that arise. There will follow a discussion of the effects such relationships between functions have on the functional models associated with a system.

In the following sections, tables are used to classify these different categories of dependent function and the notation used in these tables might warrant some explanation. The function on which the dependent function depends (so whose state triggers the dependent function, the prime function is called P and it has two elements, the trigger (t) and the effect (e), so Pt means the trigger (precondition)

of the prime function resolves to true (that function is triggered) and Pe means that the prime function's effect is present. For example, a function that is triggered when the prime function is triggered and its effect fails (maybe its purpose is to warn of failure of the prime function) can have its trigger expressed as $Pt \wedge \neg Pe$. This means, of course that the prime function's precondition (trigger) is true but its post-condition (effect) is false.

10.1 Warning or telltale functions

The first category of functions can be considered to be dependent on some other function is that of telltale (or warning) functions. The purpose of such functions is to increase the detectability of the state of the prime function with which they are associated. To this end they will have some effect that is audible or visible to the operator of the system. Possible relationships between such functions and a prime function are listed in Table 10.1. Given that a prime function is achieved when its trigger Pt and its effect Pe are true, then we might classify telltale functions by which of the prime function's conditions are used in its trigger. There are also cases where the trigger and effect of the prime function are true but the expected behaviour is not, because the effect is achieved by an alternative fault mitigating behaviour. These cases are listed in Table 10.1 together with the straightforward examples. By way of further illustration, these classes of telltale function are listed

Class	Trigger	Prime function state
input telltale	Pt	Prime function triggered
no input telltale	$\neg Pt$	Prime function not triggered
output telltale	Pe	Prime function's effect present
no output telltale	$\neg Pe$	Prime function's effect absent
behaviour warning	$\neg Pb$	Unexpected behaviour detected
confirmation	$Pt \wedge Pe$	Prime function achieved
failure warning	$Pt \wedge \neg Pe$	Prime function failed
unexpected warning	$\neg Pt \wedge Pe$	Prime function achieved unexpectedly
off confirmation	$\neg Pt \wedge \neg Pe$	Prime function inoperative
limp home warning	$Pt \wedge Pe \wedge \neg Pb$	Prime function achieved by unexpected (fault tolerant) behaviour
fault tolerant confirm	$Pt \wedge Pe \wedge Pb$	Prime function achieved by expected behaviour

Table 10.1: Telltale and warning functions

below, with examples where appropriate.

Input telltale is triggered whenever the prime function is triggered and should be achieved regardless of the post-condition of the prime function. An example is the main beam telltale on a car dashboard that lights whenever the light switches are set for main beam.

No input telltale is triggered whenever the prime function is not triggered regardless of its post-conditions. One possible example is an electrical appliance that has a telltale lamp to indicate that it is plugged in but not switched on.

Output telltale is triggered by the post-condition of the prime function, regardless of its precondition, so the prime function might be achieved unexpectedly.

No output telltale is triggered by the failure of the post-condition regardless of its precondition, so the prime function might be failed or simply off.

Behaviour warning is triggered by the failure of the behaviour (presumably according to some detectable internal state) of the prime function, so either misbehaviour is detected or the function is not active.

Confirmation This is triggered by the truth of both the pre- and post-conditions of the prime function, so it confirms that it is active and achieved correctly.

Failure warning is triggered if the trigger of the prime function is true but the effect is false, so the function has failed.

Unexpected warning is triggered if the prime function's trigger is false and the effect is true, so that function is achieved unexpectedly.

Off confirmation is triggered if both the pre-and post-conditions of the prime function are false, so the prime function is inoperative.

Limp home warning is triggered if the pre- and post-conditions of the prime function are true but there is a fault in its internal behaviour. An example would be a warning lamp showing that a fault tolerant system was running without some expected sensor data, so there is a loss of redundancy.

Fault tolerant confirm is triggered if preconditions, post-conditions and behaviour are all active as expected. It therefore confirms that a fault tolerant system is working normally, not in a fault tolerant mode. It might be used in safety critical areas in preference to a "limp home warning" as in that case a failure in the warning system might mean loss of safety in the fault

tolerant system going undetected whereas a failure of a fault tolerant confirm system will at least make it apparent that there is a failure (either in the main system or the telltale).

It should be pointed out that some of these classes of function are included for the sake of completeness. It is unlikely (but not impossible) that all of these classes will actually be found in use.

It will be seen from Table 10.1 that all combinations of a prime function's trigger and effect are covered so there is no need for any other classes of telltale function associated with failure of the prime function's behaviour provided we are willing to accept that the function operating with trigger and effect conditions satisfied implies correct behaviour and conversely that the absence of either trigger or effect implies incorrect behaviour. Given these assumptions, a telltale that the correct behaviour is taking place is identical to the confirmation function (pre- and post-conditions both present) and adding incorrect behaviour to the failure and unexpected warning functions creates conditions that will be satisfied similarly to the conditions without any express inclusion of the (internal) behaviour. Note that the use of behaviour of the prime function (as opposed to trigger or effect) as a trigger of the telltale function implies that there is some mechanism that detects the loss of some (internal) behaviour. A fault tolerant system detecting the absence of sensor data is a possible case in point.

It will be seen that where the telltale function is dependent purely on the prime function's trigger and effect, its trigger can be described in terms of the state of the prime function. If keywords describing the state of a function's trigger and effect are added (so a function is said to be `TRIGGERED` when its trigger is true regardless of the state of the effect and `EFFECTIVE` if its effect is present regardless of the state of its trigger) then this description of the trigger of a dependent function can be extended to the simple cases where only the trigger or effect is used as the trigger of the dependent function. Of course, no claims can be made for the completeness of this set of telltale functions where the conditions are not atomic. It might be the case that they are triggered either by a specific output state, such as the oven light coming on whenever the heater is on and going off when the heater is off (the oven has reached the right temperature) or because a subset of the prime function's post-conditions is sufficient to trigger the telltale function. Consider, for example, the idea of the car main beam warning being a confirmation function, so it comes on when the headlamps really are on main beam. If its purpose is to warn the driver that oncoming traffic will be dazzled, it might well be intended that it lights whenever one or both headlamps are on main beam, as one headlamp is enough to dazzle oncoming drivers even though the headlamps'

prime function is not achieved. This means that this collection of telltale functions can only be regarded as a complete set if it is sufficient to model them in terms of the state of a function, rather than any specific state of the effects of the prime function. Because the triggering of a dependent function might depend on a different combination of effects than are required to achieve the purpose of the prime function, it becomes necessary in some cases to define the trigger of the telltale function in terms of specific effects of the prime function rather than in terms of the state of the function itself. This is discussed below. This means that some way of unambiguously linking a systems' functions is required, so the correct triggers and effects are unambiguously identified. This allows the prime and dependent functions to share the mapping between the functional and system models, eliminating redundancy of description and the possible errors in mapping between the models. Approaches to this are discussed at the end of this chapter.

The significant difference between these functions and the prime function, in general, is that they are not simply triggered by a user's input to the system. This means that the trigger needs to be explicit in the functional description. While for failure analysis of prime functions (that are dependent on no other function), the triggering condition for a function can be derived from a simulation of the system behaving correctly, this is not the case if there are dependent functions. Consider a plant warning system, which lights a telltale if an expected effect of some prime function is absent. In the correct simulation, the expected effect of the prime function will never be absent, so the telltale function will not be triggered. Unless the trigger condition of the telltale function is explicitly stated in the functional description, it will not be clear whether the effect of the telltale function is expected or not. This raises the more general point that the telltale function cannot be evaluated independently of the prime function, except in the case of the simple input triggered ones. It is useless to model the output of the prime function as an external "input property" (trigger) of the dependent function as the aim of testing these functions is to show that failures of the prime function will be detected correctly. They depend on some mechanism for detecting the state of the prime function and if that function is reduced to a switch then that mechanism will not be simulated.

As the Functional Interpretation Language includes the trigger and effect of a function it is simple to incorporate either the state of the prime function or its trigger and effects in the trigger of the telltale function. For example, consider a simple warning function, being used to detect failure of, say, a ship's navigation lights.

```
FUNCTION nav_lights
  ACHIEVES ship_lit
  BY
    nav_lights_on
    TRIGGERS
    port_light AND starboard_light
```

As the failure of either navigation light needs to be detected by the telltale function, its trigger can be expressed in terms of this function.

```
FUNCTION nav_light_warning
  ACHIEVES warn_no_nav_light
  BY
    nav_lights FAILED
    TRIGGERS
    warning_lamp_lit
```

As FAILED is defined as the trigger being true and the effect false, this is identical to specifying the trigger of the warning function like this.

```
FUNCTION nav_light_warning
  ACHIEVES warn_no_nav_light
  BY
    nav_lights_on AND
      NOT (port_light AND starboard_light)
    TRIGGERS
    warning_lamp_lit
```

In this case, the description in terms of the prime function is simpler, but if a combination of trigger and effects of the prime function are used, it is possible to specify a different combination than results in achievement of the prime function. The warning lamp that either headlamp is on main beam, mentioned earlier, can be described by replacing the AND used in the main beam function with OR, so the trigger of the warning lamp's function is true if either headlamp comes on.

```
FUNCTION main_beam_warning
  ACHIEVES warn_of_dazzle
  BY
    right_main_beam OR left_main_beam
    TRIGGERS
    warning_lamp_lit
```

Here the warning function is triggered by the effects of the prime function, which case highlights the need to trigger the dependent function with the trigger and effect of the prime function, or some combination of its trigger and / or effects.

As telltale functions that warn that some unusual behaviour is causing the effect of the prime function imply that there is some means of detecting that this is the case, that can be incorporated in the warning function's trigger. This might typically be done by specifying the failure of some internal component's function (or effect). These cases are discussed in the following section on fault mitigating functions, which also depend on some internal failure being detected.

10.2 Fault mitigating functions

Another area where functions might relate closely is where the failure of a prime function is mitigated by fault tolerant behaviour. Possible cases are listed in the table in Table 10.2. These functions relate to a prime function with trigger Pt, effect Pe, normal behaviour Pb and purpose Pp. In all cases, of course, the purpose of the fault mitigation function is to enable the continued fulfilment of the purpose of the prime function, at least to a limited extent — possibly at a reduced efficiency or for a limited time. Cases where a fault mitigating function results

Class	Trigger	Effect	Purpose
backup output	$Pt \wedge \neg Pe$	backup	Pp
fault tolerant	$Pt \wedge \neg Pb$	Pe	Pp
limp home	$Pt \wedge \neg Pb$	$\subset Pe$	$\subset Pp$

Table 10.2: Fault tolerant functions

in partial fulfilment of the prime function's purpose (such as an anti lock braking system working conventionally, with no automatic anti lock control) are indicated using the subset symbol. In contrast to the functions listed in Table 10.1 above, it is possible that there are other triggers for a fault tolerant function. However, such a function will certainly share a trigger with the prime function and also depend on the loss of the required effect or of internal behaviour (such as a sensor failing). Where Pe is used in the effect column of the table, the same effectors are taken to be delivering the effect of the fault tolerant function as would be the case if the prime function was achieved. The backup effect in the top row implies the use of different effectors, such as an emergency lighting system. The $\subset$ symbol is used to indicate cases where the fault tolerant function only partly achieves the effects, and therefore partly fulfils the purpose of the prime function, such as the

loss of anti-lock brakes. The table is illustrated using textual descriptions of these classes of fault tolerant function:-

Backup In this case the purpose is (at least) partly fulfilled by some separate system that automatically switches in on failure of the main system. An emergency lighting system that lights routes to fire escapes after failure of a building's main lights is an example.

Fault tolerant Here the output of the system is identical to the prime function's output, but there is a loss of behaviour, covered by redundancy within the system. A brake by wire system continuing to work despite loss of a sensor's data is a possible example. It is likely that this will be regarded as a "limp home" mode, because the loss of redundancy means that another system failure might lead to dangerous loss of functionality.

Limp home The output of the fault tolerant function is not identical to that of the main system, but is sufficient to allow continued operation of the device. An engine management system failing in such a way that a default value for ignition timing is used instead of a calculated optimal value is an example, as is an anti-lock braking system falling back on braking conventionally following loss of sensor data.

In practice, as observed in (Manzone *et al.*, 2001), the classes of function labelled as "fault tolerant" and "limp home" are both intended for use to allow limited further use of the system. It is, of course, necessary in many areas (such as aviation and automotive) that a system will tolerate one failure for at least long enough for the user to deal safely with the consequences (such as for the aeroplane to land). The distinction drawn here is that a limp home function implies reduced functionality while a fault tolerant function allows the device to work normally (at least apparently) albeit for a restricted time and with a loss of tolerance of further failure.

It will be seen that the "limp home warning" function in Table 10.1 is identical to a confirmation function for the fault tolerant function (its precondition is that of the fault tolerant function combined with that function's postcondition using logical AND). It is likely that such a telltale will be the only sign that the system is not working normally, it being needed to warn the operator that the system requires attention before some other component failure causes the system to fail.

One difficulty with describing such functions is that the trigger of the fault mitigating function might depend on detecting some internal aspect of the system behaviour. For example, the trigger and effect of a fault mitigating function of

an engine management system in a car are apparently identical to those of the prime function, the only difference might be a loss of performance and increase in fuel consumption. However, in such systems the system itself must have some component that detects the loss of behaviour that leads to the adoption of the fault mitigating strategy, so it seem not unreasonable to incorporate that in the functional description. It is at least possible that in such cases the difference in effect between the prime and fault tolerant functionality will be ignored. For example the function of the engine might be to drive the car and the aim of the fault tolerant engine management function is to enable the engine to continue to achieve that function despite the loss of sensor data.

The example of a car's anti-lock braking system reverting to conventional braking if sensor data is missing has been mentioned. This is an example where an internal component failure will trigger the loss of the prime function and its replacement by the "limp home" function. This might be described like this.

```
FUNCTION conventional_braking
  ACHIEVES stop_vehicle
  BY
    brake_pedal_pressed AND
    (NOT right_front_rotation_data_input OR
    NOT left_front_rotation_data_input OR
    NOT right_rear_rotation_data_input OR
    NOT left_rear_rotation_data_input)
    TRIGGERS
    right_front_brake_on AND
    left_front_brake_on AND
    right_rear_brake_on AND
    left_rear_brake_on
```

This will be triggered instead of the prime anti-lock braking function, then if the anti-lock braking controller fails to receive the expected data from any of the wheel rotation sensors. There seems to be no real difficulty with this model, except for its relative complexity. The sensor data should be mapped to the system model as input to the controller module as then any failure that results in the loss of the data (whether to the sensor itself or to the network linking it to the controller) will result in the loss of the data and so the triggering of the fault tolerant function. The realistic approach to modelling these internal failures is at the component that detects the failure, not the component that causes the failure. It will be appreciated that this functionality cannot readily be modelled without a fair degree of knowledge of the likely design of the system.

One point that arises is that specification of the trigger of a dependent function in terms of function can lead to different expected behaviour compared with specifying its trigger in terms of triggers. For example, an emergency lighting system (an example of a "backup function") might be specified as being triggered when the prime function is triggered but the effect is absent. Therefore the emergency lighting function should be triggered by all the lamps in the main lighting system failing at the same time. In practice the emergency lighting function is likely to be triggered by the detection of failure of the main power supply, so the function will not actually be triggered by the failure of the lamps. Arguably, this makes the backup function independent of the prime function as the trigger and effect are both different but it does illustrate that triggers need to be chosen carefully, unless the idea is to capture the possibility that certain main system failure will not result in apparently expected behaviour of the dependent function.

It is worth noting that this discussion of fault tolerant functions excludes manual backup systems as these can be regarded as independent of the main system. An example might be a reserve parachute with its own rip cord which the parachutist will activate on finding a problem with the main canopy. This can be considered an independent system (though it clearly has a closely related purpose) except in any cases where it is desired to simulate some interlock mechanism that prevents both main and backup systems being active at once. Arguably, the enabling of the backup mechanism is then a backup function of the main system though it does, of course, not itself have any output unless some telltale is associated with it, to both warn the operator that the prime function has failed and to show that the backup can be used.

10.3 Interlocking functions

This category of dependent function differs from the others discussed here as while in the other categories, the prime function is (part of) the trigger, in this case, the state of the function upon which the interlocking function depends prevents the triggering of the interlocking function unless that function is achieved. A rather weak example is the exiting of a computer application being prevented when there is an open file that has not been saved since the last edit. This is a slightly untypical example because the likely result is a distinctive effect (a dialog box giving the user the chance to save the open file) rather than simply preventing the closing of the application. In a hardware system, a more likely case is that the dependent function is simply prevented by the non-achievement of the function upon which it depends. A better example (if less familiar) is a railway signalling

system, where the interlocking system makes it impossible to pull the levers (in a old fashioned mechanical system) to clear the signals for a given route unless the points have already been set correctly for that route. A related example might be the accessories in a car that will not operate unless the ignition has been switched to the right position, so power is available. This differs from the signalling example in that the driver can switch on the windscreen wipers (say) before starting the car, but they won't come on until the car is started (or at least the process of starting the ignition is in progress). This case does differ in that the ignition switch simply gets added to the trigger for the wiper circuit, whereas in both the unsaved work and signalling examples, the relationship between the functions is more complex.

The signalling system is simpler to model as if the interlocking function is blocked, nothing will happen. Given the setting of the points concerned is described in a `route_set` function, a signalling function can be described like this.

```
FUNCTION pull_off_signals
  ACHIEVES road_shown_clear
  BY
    FUNCTION route_set ACHIEVED AND pull_lever
    TRIGGERS
    signal_cleared
```

Naturally in this case, the route and signals concerned would need to be specified, but it is suggested that this simple example illustrates the point that the trigger of the interlocking function depends on achievement of another function, that upon which the interlocking function depends.

The unsaved work example is more complicated as here there will be an alternative effect if the user attempts to close the application if there is unsaved work that will be lost. The simplest approach is to have two functions in this case. The closing function might look similar to the signalling example above, though in practice it might be better associated with the system state that is the effect of the save function.

```
FUNCTION close_application
  ACHIEVES work_over
  BY
    clicking_exit AND no_unsaved_work
    TRIGGERS
    application_closes
```

This is complicated slightly as while the effect `no_unsaved_work` is a result of achievement of the save function, it is also true if no changes have been made to any file since opening the application. This illustrates the need for the functional language to be able to have effects (or states) in common with several functions, which is the most significant effect that the idea of dependent functions has on the Functional Interpretation Language. While the functional description above serves to show that the application should not close if there is unsaved work, it does nothing to specify what should happen if the user tries to close the application and there is unsaved work open. This can be done with another function.

```
FUNCTION save_reminder
  ACHIEVES warn_of_unsaved_work
  BY
    clicking_exit AND NOT no_unsaved_work
    TRIGGERS
    FUNCTION save_option_dialog
```

So all clicking the exit button achieves itself is the display of the options dialog giving the user the chance to save the open work. This is itself a separate function.

```
FUNCTION save_option_dialog
  ACHIEVES forces_exit
  BY
    FUNCTION save_and_close
    OR
    FUNCTION close_losing_changes
```

This leads to yet another pair of related functions, of which `save_and_close` might be described as follows.

```
FUNCTION save_and_close
  ACHIEVES work_over
  BY
    clicking_save_option
    TRIGGERS
    no_unsaved_work AND application_closes
```

It might well be the case that there is no need for the `save_option_dialog` function, indeed it is arguable whether it is really a separate function as opposed to the means by which the `save_and_close` and `close_losing_changes` functions are made usable. The use of this set of functions depends in how detailed a

functional model is required, to what extent the means of accessing the closing functions needs specifying in the functional model.

In the example, it is assumed that names are unique and can be used to match functions and effects unambiguously, as well as matching descriptions of purpose (which separate models). This suggests that there is a case for separating all elements of the functional description, or at least for allowing one function to unambiguously refer to another function's trigger or effect. This both reduces the work in carrying out the mapping between the trigger and effect and the system model, as there is no need to reproduce the same mapping for each function. This also reduces the danger of mappings that should match not doing so following an error in making the mappings. This is discussed in Section 10.5 below. Another point worth making in passing is that the functional model (or that part of it illustrated here) of the software example might seem complex, but that is (arguably) an inevitable result of the complexity of intended behaviour of many software systems. Indeed, the need for a functional interpretation language to model such systems was one of the motivations for the present work.

10.4 Recharging functions

While there are various examples of systems that include a recharging function, the purpose is similar in all cases, being to return the system to a state where it is ready for a repeat of the prime function. Examples include the refilling of a cistern after flushing a W. C., motorised winding on of the film and cocking of the shutter of a motor driven camera and reloading a gun. A failure of the recharging function means that the prime function can not be achieved subsequently to the failure of the recharging function. These functions appear to share a trigger with the prime function, so pressing the shutter release causes the camera to both expose a frame and to wind on to the next. This can be contrasted with a camera with a manual wind where the two functions are clearly separate. This example does perhaps also suggest that the recharging function is to be considered a separate function even when it does share a trigger with the prime function. The purpose (and consequences of failure) are different for the prime and recharging functions. If the prime function fails then the purpose of the system is unfulfilled (the photo has been missed, for example) while when the recharging function fails, at least at first, the main function is achieved (so the photo was taken) but the system is not ready for the next use, the next photo will be missed. This assumes that before the failure of the recharging function, the system was in a state ready for the achievement of the prime function which it should be if the recharging function was previously working correctly. It would be possible to use subsidiary functions to

model this case, combining the two functions under a top level "system working" function, like this.

```
FUNCTION take_photo
  ACHIEVES photograph_taken
  BY
    press_button
    TRIGGERS
    PIF expose_frame
    L-SEQ
    PIF wind_on
```

As the two desired effects are associated with their own subsidiary functions, each failure can be associated with its own consequence, so the `expose_frame` function might be described like this.

```
PIF expose_frame
  ACHIEVES frame_exposed
  BY
    shutter_open L-SEQ shutter_closed
```

It will be seen that there is a difficulty here as the purpose of the `expose_frame` function is essentially the same as that of the `take_photograph` function. It is at least arguable that in this case, the top level function does not really have a single clearly defined purpose, but rather a combination of the purposes associated with the two subsidiary functions.

A more practical objection to this approach is that the two subsidiary functions do not really share a trigger. The `wind_on` function is actually triggered by the achievement of the `expose_frame` function. If that function fails then there is no call for the camera to wind on. If the functional description treats both as sharing a trigger, then can the failure of one or both be distinguished? After all, if the shutter (say) fails then the camera might well not wind on either. This could, perhaps, be worked around by modelling the recharging function in terms of the state of the system after the achievement of the function, so the shutter is cocked and an unexposed frame is behind the shutter, rather than the expected behaviour, the action of winding on. A related difficulty, though, is that in this case, the unexpected achievement of the recharging function will be missed if, say, the film winds on despite no frame being exposed when the shutter release was pressed. If the recharging function is considered purely in terms of the goal state (as suggested above) then the fact that a frame is wasted (to continue with

the camera example) is missed. Indeed, the fact that the recharging function is achieved unexpectedly is missed because if it is described as sharing a trigger with the prime function (pressing the shutter release) then it is apparently achieved, both trigger and effect are true. If it is described as triggered by the achievement of the prime function then if that function fails, the recharging function is correctly shown as being unexpected.

It is therefore suggested that, while the approach described above might give an adequate model in some cases, it is more correct to regard the recharging function as a dependent function of the prime function.

This can simply be done by describing the recharging function with the trigger and effect of the prime function (or even just its effect) as the trigger of the recharging function. Whether the prime function's trigger is included will depend on what the recharging function is expected to do if the prime function is achieved unexpectedly (so was not triggered). In general, it is likely that the recharging function will be expected in this case, but it might not be if there are safety issues. For example, it might be intended that a gun will not automatically reload following an accidental discharge. The pair of functions for the motorised camera might be described like this.

```
FUNCTION expose_frame
  ACHIEVES photograph_taken
  BY
    press_shutter_release
    TRIGGERS
    shutter_open L-SEQ shutter_closed
```

and

```
FUNCTION wind_on
  ACHIEVES film_wound
  BY
    FUNCTION expose_frame
    TRIGGERS
    film_transport AND shutter_cocked
```

This is arguably a rather simplistic attempt at a functional description of a motorised camera. It does imply that if the shutter trips unexpectedly (an admittedly unlikely failure) the film should not wind on. Arguably, of course, if the shutter tripping was unexpected, so was the winding on and also, since the frame previously in the gate might have been exposed, the fact that the film is

wound on is a benefit. A worse difficulty is that the `expose_frame` function might actually require more effects than in the model above. If the camera is a single lens reflex, the mirror will be moved and the lens diaphragm adjusted. How these effects are incorporated into the `expose_frame` function will affect the trigger of the `wind_on` function. It is likely that its trigger will be the effect of the shutter opening and closing, even though the failure of one of these other effects might result in failure of the `expose_frame` function. There is no reason why this cannot be described using the functional language, but it does emphasis the point that a complex system might require a complex functional description. It is suggested, however, that this does indicate a rôle for the functional language in refining a proposed system design, if only by ensuring that problems like that mentioned are considered as part of the functional design of the system. Thus is discussed further in the chapter on evaluation of the functional language, Chapter 11.

One area that does vary from system to system is that some systems might allow a sequence of repetitions of the pair of functions (such as a photographer shooting a series of photographs by keeping the shutter release pressed) and others allow no such repetition, such as the W. C. example. These cases can be described using a cycle of the prime and recharging functions. For example, suppose the camera is intended to shoot a sequence of photographs if the shutter release is held down.

```
FUNCTION shoot_sequence
  ACHIEVES photographs_taken
  BY
    shutter_button_pressed
    TRIGGERS
    CYCLE
    PIF expose_frame
    L-SEQ
    PIF wound_on
    NEW_CYCLE
```

This is similar to the simple case described above, so similar difficulties apply. An additional problem is that strictly, the achievement of each function triggers the other. If, say, the `wound_on` function fails, it is not the case that the shutter will continue to trip repeatedly. This is a further example of the ambiguity introduced by having the two functions share a trigger. These complexities are discussed further in the following chapter, where the functional language is evaluated by describing some case studies.

Having discussed possible categories of dependent function and how they can be described using the Functional Interpretation Language, some matters that arise can be now discussed.

10.5 Relations between functions

The examples discussed above highlight the need for functions to be able to refer to elements of some other function associated with the same system. For example, all but one of the functions in the closing application example discussed earlier refer to the effect labelled as `no_unsaved_work`. Strictly speaking, of course, this is a state of the system, which is either true of false, and the effect (of the save function) is to make that state true. This sharing of elements of function leads to the need to ensure that these elements are unambiguously identified. The approach suggested is to associate an element with its (original) parent function, separating them with a dot. So if the `no_unsaved_work` effect's truth is associated with the save function, other functions can refer to the element unambiguously as `save.no_unsaved_work`. This notation is similar to that used to refer to a function's trigger or effect in mapping between the functional and system models. All such qualified instances of the element are then mapped to the system model together, ensuring the mapping is identical. In addition functions are labelled as such (as are incomplete functions as PIFs, OIFs and TIFs) to indicate both the need to look up that part of a system's functional model and to specify that the label is not to be mapped to a system property, which would be needed if it is a trigger or effect. These labels have already been used in the discussion of functional decomposition in Chapter 6.

Rather than specifying the prime function's trigger and effect in the trigger of a dependent function, this might be specified in terms of the prime function's state. This does depend on the trigger for the prime function being identical to the combination of trigger and effects associated with the appropriate state of the prime function. The use of `TRIGGERED` and `EFFECTIVE` allows this even in the case of simple telltales. To give a simple example, a car's main beam warning lamp's function can have the trigger `FUNCTION main_beam TRIGGERED` which simply means that the telltale lamp's function shares the trigger expression with the indicated function. The keyword `EFFECTIVE` is used to show that a function is triggered by the truth of some other function's effect expression. This avoids the need to repeat the possibly complex conditions associated with the prime functions trigger (or effect).

All these relationships are between function associated with one system, of course, as there is nothing to be gained by specifying relationships with a system that will not be simulated. If the prime function's system is not simulated, there will then be no way of finding whether the dependent function's triggering conditions have been met. This leads to the idea that there is a fuller set of rela-

tionships between functions than the decompositions in Chapter 6 but this fuller set incorporates the decompositional relations. It will be seen that the set of relations in a functional decomposition is defined by a specific purpose, while that of this more general functional relationship is defined by the system with which they are associated. This system might, of course, be intended to fulfil several distinct purposes so the general set of system functions might include several functional decompositions as well as relations expressing other functional dependencies.

One interesting area of relations between functions is where a warning function affects the detection value of the consequences of failure of a description of purpose associated with a prime function. An approach to describing this relation is to replace the detection value in the consequences of failure in the telltale function's description of purpose (which will refer to that of the prime function) with a revised (higher) value for the prime function's detection. For example, a warning function's description of purpose might be written like this.

```
PURPOSE warn_of_failure
  DESCRIPTION "Warn user system has failed"
  FAILURE "User not warned of system failure"
    SEVERITY 3
    DETECTION PURPOSE system_purpose.FAILURE.DETECTION 9
```

Here, `system_purpose` is the description of purpose associated with the prime function and so its value for detection is then changed to the specified value rather than the (presumably) lower value in that description. A similar approach can be used if the telltale function is triggered by the presence rather than the absence of the prime function's effects, only now the changed detection value might be that for unexpected achievement of an effect (or a function, if its description of purpose includes consequences of unexpected achievement). This approach to describing description of telltale and warning functions is, of course, available as an alternative to the usual use of the detection value. The usual approach can be used in those cases where it is felt more appropriate.

10.6 Simulation of dependent functionality

One aspect of the modelling of these dependent functions is that many of them are triggered by failure in a system. This means that the triggering of these functions cannot be derived from the simulation of the system's behaviour with no component failure in failure analysis as the function will not be triggered in that simulation. This has already been mentioned and is a reason for including

the trigger of a function in its description, even though it might be considered unnecessary for failure analysis.

Another result of this dependence on failure in the simulation of the system is that simulations of the system with component failure are required for design verification of the subsystems that implement these dependent functions. Clearly the functioning of a fault tolerant system function will not be tested without testing the whole system with some failure that should result in the loss of the prime function. Indeed, the incorporation of such dependent functions can be seen as blurring the distinction between design verification and failure analysis. Another aspect of this is that failures that cause the dependent function to fail (in failure analysis) will only be detected if the system is simulated with some failure that would otherwise trigger the dependent function, in addition to the failure in the subsystem that implements the dependent function.

A point that arises from this is that in general a subsystem that implements a dependent function can be specified in terms of the state (or triggers and effects) of the prime function, but it will in many cases actually be triggered by some detection device within the system. This leads to the possibility that the actual effects of a prime function are absent, but this is undetected by the device that triggers the dependent function. This might result from careless design so if the warning lamp in the navigation lights example mentioned earlier was triggered by the presence of current in some part of the circuit common to both navigation lamps (to obviate the need for two detectors) then, of course, the failure of one lamp will be missed as current will still flow in that part of the circuit. If the warning function is described in terms of the effects of the prime function, this failure of the warning function will be detected on running a failure analysis of the system and will be included in the resulting design analysis report. The design analysis system can distinguish those system failures that cause both the prime and dependent functions to fail from those that cause the failure of only one or the other. The navigation lamp example with one detector, as described above illustrates this, as a failure that causes one navigation lamp to fail will also cause the warning system to fail, if it is defined in terms of the prime function. Notice that if one lamp fails the warning function is triggered (the prime function has failed) but it too will fail (its triggering condition is true and it effect absent) so this can be included in the design analysis report, highlighting the inadequacy of the design of the warning system.

10.7 Uses of modelling of dependent functions

The uses of modelling dependent functions have been touched upon in passing throughout this chapter, but it seems worth finishing with a summary of these.

Clearly, if a system that incorporates dependent functions is to be described accurately, then these functions need to be described accurately in turn. This depends on the subsystems that implement these dependent functions being described alongside and simulated as part of the system that implements the prime function on which the function depends. It has already been noted that the idea of defining a dependent function in terms of the prime function on which it depends is interesting for design verification as it can be used to highlight those system failures that result in the dependent function not triggering even though its triggering condition is true. This was illustrated using the navigation light example in the previous section so will not be repeated here.

Another possible use of the Functional Interpretation Language for modelling dependent functions (though it applies to other functions as well) is that it could be used, independently of its use for interpretation of simulation, for refining the design of such functionality. For example, a dependent function might (initially) be defined in terms of the prime function (so a failure warning function is triggered whenever the state of the prime function is failed) but might be refined to depend on the actual trigger and effects of the prime function. Initially defining a dependent function in terms of the state of the related prime function could highlight cases where the implementation of the dependent function fails to achieve the intended purpose, so it is not triggered when it was expected for example. Indeed, it would even be possible to describe the dependent function twice, once in terms of the state of the prime function and once in terms of the dependent function's actual trigger to emphasis such mismatches as part of the process of design verification of the system that implements the dependent function.

One point that emerges from the foregoing discussion of dependent functions is that they illustrate the distinction to be drawn between the functional and physical decompositions of some product. The system that implements a warning function, for example, might be considered quite distinct from the system that implements the prime function (though clearly some connection will be required) but the functionality of the product as a whole depends on the correct functional relationship between these systems so even though they might be structurally separate, they are functionally related. An example is the incorporation of a dashboard warning lamp into the instrumentation system on the structural side and the exterior lighting system on the functional side. This does mean that for

correct design analysis of dependent functions, the modelling of the system must follow the functional decomposition of the product as a whole.

Having considered the uses of the Functional Interpretation Language in design analysis of these dependent functions, the use of the language in general can now be discussed.

Chapter 11

The Functional Interpretation Language in use

While earlier chapters have made use of examples to illustrate the discussions therein, these examples have been somewhat fragmentary in nature and it seems well worth illustrating the use of the language with some more fully explored case studies. There are two primary motives for this. The first is to evaluate the use of the language in design analysis of systems of realistic size and complexity. The other motivation is that this offers the opportunity to combine the aspects of the work that have been described and discussed in different chapters of the thesis, drawing together the different areas of the language, so as to give a better overview of the language as a whole. The evaluation will, naturally, be done primarily in terms of the language's intended use for interpretation and automation of design analysis. However, the use of the language in related tasks (such as diagnosis) will be discussed as will the possibility of making use of the language in refining the functional specification of a system as part of the design process. Finally, the chapter will consider the relationship between the functional language and the simulation engine (or engines) used to generate the simulation whose interpretation the language is to enable.

One difficulty in selecting suitable example systems is that such a system needs to be complex enough to support the arguments made, while being simple enough for these arguments to be readily followed. It is hoped that the examples chosen are appropriate. They are mostly familiar examples, so an informal grasp of their functionality should be easily acquired.

11.1 The Functional Interpretation Language and design analysis

This section will explore the use of the language in more depth than was done in the illustrative examples in the text, by describing how the language might be used to describe the required functions of a relatively simple real-world system as well as discussing how the use of the language works in concert with a model based (or indeed a mathematical) simulation of the system under analysis. The use of the Functional Interpretation Language for interpretation of simulation can be illustrated by a system from one of the industrial partners on the SoftFMEA project, under the auspices of which much of the present work was carried out. This is a system that warns the driver of a car when s/he has not buckled the seat belt, or the front seat passenger (if any) has not done so. It does this by lighting a dashboard lamp and intermittently sounding a "chimer" for a period, before the chimer falls silent. This system has been used in fragmentary examples earlier in the thesis, but it will now be discussed in more detail.

11.1.1 Seat belt warning system

This is an example of an electrical system of the type that the design analysis tool developed in earlier work in Aberystwyth (*AutoSteve*) was intended to model. However, its behaviour proved too complex for the existing design analysis tool and, indeed, the failure to model this system correctly was one of the motivations for starting on the work described in this thesis. The schematic circuit diagram is shown in Figure 11.1. This schematic was redrawn from one supplied by one of the SoftFMEA project's industrial partners. It was changed because the original system included other functionality. The electronic control unit (ECU) labelled "RCU" (the Restraint Control Unit) is also responsible for controlling the car's airbags. It was part of the design brief for the original seat belt warning system that it should use existing components where possible. The schematic in Figure 11.1 separates out the parts of the original schematic that are concerned with the seat belt warning system. For simplicity, only those components referred to in the text are named. The actual schematic will identify all components and might be slightly complicated by including connectors and several wires in series where a run is so divided. The connecting pins have been labelled to allow direction and flow of current to be identified. It is, perhaps, worth observing that all the connections are electrical, rather than being network bus connections, for example so the system can be simulated either by a qualitative circuit analysis tool, such

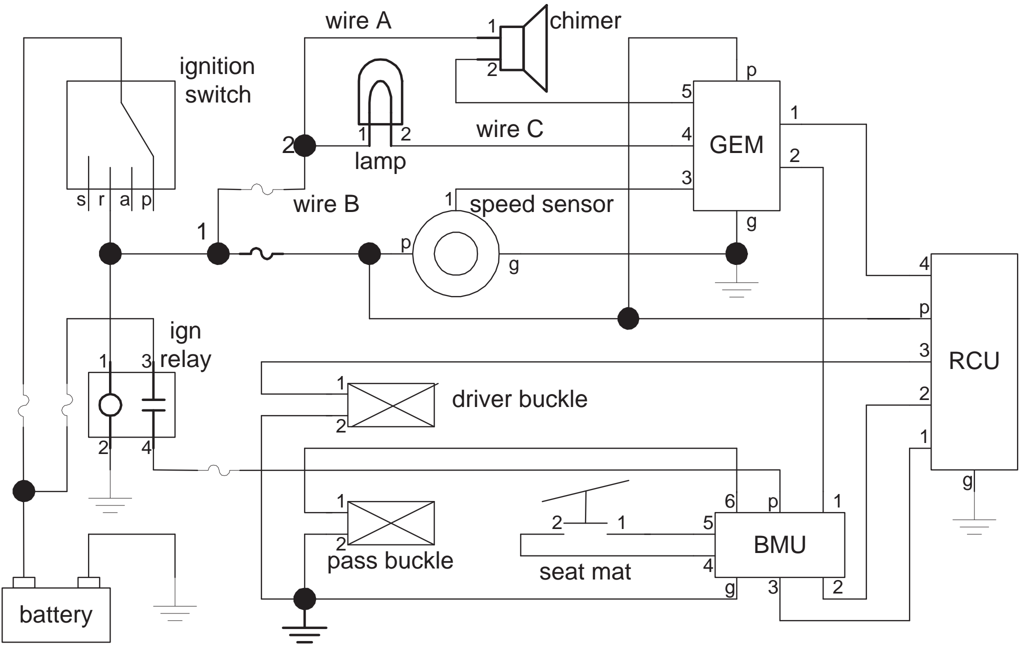


Figure 11.1: Schematic for the seat belt warning system.

as CIRQ, (Lee, 1999a), or a numerical tool, such as Saber (Saber, 1996). Such a simulation is briefly described in the following section.

One shortcoming of the way this schematic has been drawn is that the speed sensor does not have the behaviour of its input. It is essentially treated as a switch with two possible states, one for when the car is travelling below the threshold speed and one for when it is travelling at or above that speed. The speed is simply set manually in running the simulation. Ideally, some way of modelling the actual behaviour of the speed sensor would be incorporated, but this would further complicate the schematic and it was not included in the industrial partner's original schematic from which the belt warning system schematic was drawn.

The main function of the system is, of course, to warn the driver that his or her seat belt in unbuckled and to give a similar warning if the front seat is occupied by an unbuckled passenger. In addition there are two other functions (in the sense of intended behaviours) of this system that can be informally stated as follows:-

- The system is to be capable of being temporarily disabled, so the car can be manoeuvred without the system being activated. This temporary disabling to last until next time the engine is stopped and re-started.
- The system can be permanently disabled. Strictly speaking this means disabled until the disabling is manually overridden whereupon the system should function normally.

The functional modelling of this system was actually derived from knowledge of system's (intended) behaviour, expressed in state charts describing the system. These showed that the trigger for the temporary disabling of the seat belt warning function is the buckling and unbuckling of the driver's seat belt within a fixed time and the indefinite disabling was triggered by repeated buckling and unbuckling of the driver's seat belt, five repetitions being required. How these can be described in the Functional Interpretation language and how this is used to interpret the simulation is illustrated in Section 11.1.3.

11.1.2 Simulation of the seat belt warning system

The Functional Interpretation Language is intended to be independent of the simulator used, in that the functional models can be mapped to system properties associated with either a qualitative or a numerical simulator, for example. However, for the purpose of this illustration, it does seem worth briefly describing the simulation of the example system. The seat belt warning system is electrical but with significant software components, the ECUs. For the sake of this illustration, a multiple level qualitative circuit simulator such as MCIRQ, (Lee *et al.*, 2001) is used, combined with state chart models of the more complex components (the ECUs), as described in (Snooke, 1999). However, to save space in the description, the behaviour of the ECUs will be described without state charts.

The qualitative circuit simulator uses four levels of resistance in this case, INF, HIGH, LOW and ZERO where wires and closed switches have the value ZERO, components such as the lamp and chimer have the value LOW and the internal resistance of solid state components (such as the ECUs) is modelled as HIGH. Open switches are treated as being INF. Each path from power to ground has as its overall level of resistance that of the highest component resistance and this is associated with a level of current of NONE, LOW, ACTIVE and SHORT. To simplify component models, when the current through the lamp is ACTIVE it is taken to be lit and likewise when the current through the chimer is ACTIVE, it is sounding. Control of the intermittent chiming is managed by the GEM. Each switch has positions associated with a different level of resistance. The "driver buckle" and "pass buckle" have positions labelled buckled (associated with a resistance of INF) and unbuckled, when the resistance is ZERO. The "seat mat" has positions occupied (ZERO resistance) and unoccupied (INF resistance). The values for these switches are chosen because if they were reversed any breakages in the wires connecting those switches could result in false warnings. As they are, such failures disable the system, avoiding potentially distracting behaviour. The

ignition switch has a resistance of INF in positions ‘p’, ‘a’ (accessories) and ‘s’ (start) and a resistance of zero in ‘r’ (run). This means that the whole system is only active when the engine is (or should be) running. The switch that is between pins 3 and 4 in the “ign relay” has a resistance of zero when the current through the coil, between terminals 1 and 2 is ACTIVE.

As MCIRQ models circuit behaviour in terms of resistance and current, the behaviour of the ECUs has to be described in these terms. This would be done using state charts, but as this illustration is not concerned with the disabling functions (which entail one of the ECUs having some internal memory) textual descriptions will suffice. These are described using dependency expressions in terms of current i and resistance r rather than voltage as this is what the simulator supports. In all these cases, the resistance that is dependent on the current is INF unless the current expression is true. The internal resistances are named after the pins they can be taken to join, so $(p, 1)$ joins the pin connected to power to the pin numbered 1. Currents are indicated by $\rightarrow$ so $(p \rightarrow 1)$ means current flowing from p to 1. In each case, the expressions are local to the component concerned, named in the commentary so the name has been omitted from the expression.

The BMU (belt minder unit) passes a current to the pin connecting it with the GEM if there is LOW current between its power pin and the pass buckle pin and if there is LOW current between the two seat mat pins, indicating that the seat is occupied but the seat belt not buckled.

$$r(p, 1) = \text{ HIGH if } i(p \rightarrow 5) == \text{LOW} \wedge i(p \rightarrow 6) == \text{LOW}$$

The RCU passes current to the GEM if there is LOW current between its power pin and the driver buckle pin.

$$r(p, 4) = \text{ HIGH if } i(p \rightarrow 3) == \text{LOW}$$

The speed sensor is treated as a switch with the states fast (zero resistance between power and the GEM pin) and slow (INF resistance).

The GEM allows current to flow between its lamp pin and ground if there is current between the sensor pin and ground and between either the RCU pin and ground or the belt minder unit pin and ground (or both).

$$r(4, g) = \text{ ZERO if } i(3 \rightarrow g) == \text{LOW} \wedge (i(1 \rightarrow g) == \text{LOW} \vee i(2 \rightarrow g) == \text{LOW})$$

When the GEM enters that state it will also reduce the resistance between the chimer pin and ground to ZERO for six intervals of 600 milliseconds, separated

by interruptions of 300 milliseconds. The use of current rather than voltage is a limitation of the simulator used.

The simulation can be run using a sequence of circuit states, such as switch positions. As most of these give uninteresting results, the sample output listed below is from one specific state. It also is run on the circuit operating correctly, with no component failures. The output is further simplified by only listing the current values for those components named on the schematic. In practice it would list values for each resistance in the circuit, including those internal to the ECUs. It is hoped that this representation of the output is sufficiently detailed despite these simplifications. This sample lists the output of the simulator the engine running, the car going fast, the driver buckled and the passenger unbuckled and also that the change from the previous simulation is that the car is now going fast (so the speed sensor state has changed).

```
Input system state:
  ignition switch.position = run
  driver buckle.position = buckled
  pass buckle.position = unbuckled
  seat mat.position = occupied
  speed sensor.state = fast
Result: step 1
  ign relay 1 to 2 current = ACTIVE
  ign relay 3 to 4 current = LOW
  lamp current = ZERO
  chimer current = ZERO
  driver buckle current 1 to 2 = ZERO
  pass buckle 1 to 2 current = LOW
  seat mat 1 to 2 current = LOW
  wire A current = ZERO
  wire B current = ZERO
  wire C current = ZERO
  BMU p to 1 current = LOW
  BMU p to 2 current = LOW
  BMU p to 3 current = LOW
  BMU p to 5 current = LOW
  BMU P to 6 current = LOW
  BMU 4 to g current = LOW
  RCU p to 3 current = ZERO
  RCU p to 4 current = ZERO
  RCU 1 to g current = LOW
  RCU 2 to g current = LOW
  speed sensor p to 1 current = low
  GEM 1 to g current = ZERO
  GEM 2 to g current = LOW
  GEM 3 to g current = LOW
  GEM 4 to g current = ZERO
  GEM 5 to g current = ZERO
```

```
step 2: current LOW in GEM 3 to g changes resistance
of GEM 4,g and GEM 5,g to ZERO.
Changes are:
  lamp 1 to 2 current = ACTIVE
  chimer 1 to 2 current = ACTIVE
  wire A splice 2 to chimer current = ACTIVE
  wire B splice 1 to splice 2 current = ACTIVE
  wire C lamp to GEM current = ACTIVE
  GEM 4 to g current = ACTIVE
  GEM 5 to g current = ACTIVE
step 3: after 600mS, GEM changes resistance of GEM 5,g to INF.
Changes are:
  chimer current = ZERO
  wire A current = ZERO
  GEM 5,g current = ZERO
step 4: after 300mS, GEM changes resistance of GEM 5,g to ZERO.
Changes are:
  chimer 1 to 2 current = ACTIVE
  wire A splice 2 to chimer current = ACTIVE
  GEM 5 TO g current = ACTIVE
step 5: after 600mS, GEM changes resistance of GEM 5,g to INF.
Changes are:
  chimer current = ZERO
  wire A current = ZERO
  GEM 5,g current = ZERO
step 6: after 300mS, GEM changes resistance of GEM 5,g to ZERO.
Changes are:
  chimer 1 to 2 current = ACTIVE
  wire A splice 2 to chimer current = ACTIVE
  GEM 5 TO g current = ACTIVE
step 7: after 600mS, GEM changes resistance of GEM 5,g to INF.
Changes are:
  chimer current = ZERO
  wire A current = ZERO
  GEM 5,g current = ZERO
step 8: after 300mS, GEM changes resistance of GEM 5,g to ZERO.
Changes are:
  chimer 1 to 2 current = ACTIVE
  wire A splice 2 to chimer current = ACTIVE
  GEM 5 TO g current = ACTIVE
step 9: after 600mS, GEM changes resistance of GEM 5,g to INF.
Changes are:
  chimer current = ZERO
  wire A current = ZERO
  GEM 5,g current = ZERO
step 10: after 300mS, GEM changes resistance of GEM 5,g to ZERO.
Changes are:
  chimer 1 to 2 current = ACTIVE
  wire A splice 2 to chimer current = ACTIVE
  GEM 5 TO g current = ACTIVE
step 11: after 600mS, GEM changes resistance of GEM 5,g to INF.
```

```
Changes are:
  chimer current = ZERO
  wire A current = ZERO
  GEM 5,g current = ZERO
step 12: after 300mS, GEM changes resistance of GEM 5,g to ZERO.
Changes are:
  chimer 1 to 2 current = ACTIVE
  wire A splice 2 to chimer current = ACTIVE
  GEM 5 TO g current = ACTIVE
step 13: after 600mS, GEM changes resistance of GEM 5,g to INF.
Changes are:
  chimer current = ZERO
  wire A current = ZERO
  GEM 5,g current = ZERO
```

The simulation works by running the circuit simulator on the circuit in the state corresponding to the input settings and then checking the component behavioural description to see if any component's internal state changes. In this case, the current flowing in the GEM from the speed sensor results in the lamp's path to ground being completed and the intermittent chiming sequence being started. This result in current starting to flow through the lamp and chimer. Successive steps simply alternate the resistance of the chimer's path to ground between ZERO and INF, according to the timing of the GEM's behaviour.

It will be seen that despite omitting many of the components (the wires) and the reasonable simplifying assumption that only those current values that change between steps are included in the next step, the output of the simulator is substantial. This extract is for one switch setting, and with no failures. For an FMEA, the output will include a sequence of switch settings repeated for each failure mode as well as the correct simulation. Even with this relatively simple system, it will run to several hundred pages. This illustrates the value of interpreting the output so only the significant results are shown.

Despite the advantages of combining the simulation with the interpretation enabled by the use of the FIL, there are occasions when running the simulation alone is sufficient, such as checking the behaviour of the system in one specific state. This could be illustrated by colouring the schematic to show the current flows.

Having described a suitable approach to simulation of the example system, it is time to consider the functional modelling needed to allow interpretation of these results.

11.1.3 Functional description of the seat belt warning system

The main function of the this system has already been used to illustrate various features of the Functional Interpretation Language. It might be described as follows.

```
FUNCTION give_warning
  ACHIEVES warn_unbuckled
  BY
    speed > threshold
    AND
    (driver_unbuckled
    OR
    (passenger_present AND passenger_unbuckled))
    TRIGGERS
    PIF show_warning_lamp
    AND
    PIF sound_chimer
```

Strictly speaking, of course, this does not capture the condition that the ignition is on, though it might be argued that since the car is moving, the ignition will be on (unless the car is rolling down hill, of course). The trigger is a moderately complex expression because the triggering of the warning function does depend on a variety of factors. It will be noted that the factors given here do not include the possibility of disabling the warning system. This will be discussed later. It will also be seen that the effects associated with the function have been labelled as purposive incomplete functions (PIFs) as each can be regarded as fulfilling its own aspect of the purpose of the function or, pragmatically, that the failure of either one of the effects is less serious than the failure of both, as discussed in Chapter 7. It is worth noting that the function need not have been described in this way. If the designer who was creating the functional description felt that the consequences of the loss of either one of the effects were as serious as those of the loss of both, the effects could have been described as such, rather than as subsidiary functions.

The purpose the `give_warning` function fulfils might be described like this.

```
PURPOSE warn_unbuckled
  DESCRIPTION "Warns driver that seat belt needs buckling."
  FAILURE_CONSEQUENCE "Driver not warned of unbuckled seat belt
                      Risk of injury."
```

```
    SEVERITY 4
    DETECTION 7
```

As was discussed in Chapters 6 and 7, these consequences will be included in the design analysis report if the trigger results in neither effect (PIF) being achieved. In the case of a failure analysis (such as FMEA), this might result from wire B breaking (going open circuit) so supply to both the chimer and the lamp is interrupted.

As they are PIFs, there are separate functional descriptions for the effects of the `give_warning` function. The `show_warning_lamp` might look like this.

```
PIF show_warning_lamp
  ACHIEVES visual_unbuckled_warning
  BY
    lamp_lit
```

This quite simple. Being a PIF it has no trigger expression of its own, it "inherits" the one from the top level function, which is shared with the other PIF. As a subsidiary function, `show_warning_lamp` is associated with its own description of purpose.

```
PURPOSE visual_unbuckled_warning
  DESCRIPTION "Visually warns driver that seat belt needs buckling."
  FAILURE_CONSEQUENCE "No persistent warning of unbuckled seat belt
                      Risk of injury."
    SEVERITY 3
    DETECTION 4
```

As this description of purpose is only used if the other subsidiary function is present, it can be described making that assumption and it is also acceptable to draw on some knowledge of the nature of the other associated effect (the other subsidiary function). In this case, it is known that the `sound_chimer` subsidiary function is not persistent, so if the driver is concentrating on the manoeuvre being carried out so that s/he fails to hear the chimer, no warning will be noticed. It is suggested that the presence of the chimer both reduces the severity and detection values for the failure of this subsidiary function because the sounding of the chimer will draw attention both to the triggering of the main function and to the absence of the expected dashboard lamp.

The `sound_chimer` subsidiary function is a good deal more complex, depending as it does on an effect that is intermittent in nature and that terminates in a

passive state (with the chimer silent). For this reason, this function has been used as an example in Chapter 8. In that chapter, a simplified version of the sounding function was used, in which a fixed number of "chimes" were counted. In the actual system, the behaviour was governed by timing, so the chimer sounds for a set period of time and is silent for a set period of time and this alternation continues for a set period of time, whereupon the chimer remains silent. There were similar examples in Chapter 9. The correct version can be described like this.

```
PIF sound_chimer
  ACHIEVES audible_unbuckled_warning
  BY
    CYCLE [BEFORE 250ms]
    SEQ chimer_on [AFTER 250ms BEFORE 350ms]
    SEQ NOT chimer_on [AFTER 500ms BEFORE 750ms]
    NEW_CYCLE
    SEQ NOT chimer_on [AFTER 5s BEFORE 7s]
```

This is, unfortunately, an example of the Functional Interpretation Language that does not read easily, as was discussed in Chapter 9. The intended behaviour is that the cycle of chimer on and chimer off starts with the chimer coming on within 250 milliseconds of the trigger of the top level function becoming true and each sound of the chimer should last for a period of at least 500 milliseconds but no more than 750, before a silence of between 250 and 350 milliseconds. Finally, the chimer should fall silent and remain silent after at least 5 seconds of the chiming sequence but after no more than 7 seconds. This means that there will be at least five chimes but there might be as many as nine (though in that case, the final one should be cut short). This functional description is written with the assumption that the chimer can be regarded as being either sounding or silent. As the `show_warning_lamp` function has no temporal constraint, the lamp can come on at any time before, during or after the chiming sequence. As the two PIFs share a trigger, if any temporal relationship between these functions was required, this can be achieved as each timing sequence will start from the same triggering event. One point that arises from this description is that the functional description need not specify which component is responsible for the timing of the chimer sequence. It might be governed by the GEM or by some timing device in the horn itself.

The `audible_unbuckled_warning` description of purpose is very similar to the `visual_unbuckled_warning` one associated with the lamp subsidiary function, but it is given here for the sake of completeness.

```
PURPOSE audible_unbuckled_warning
  DESCRIPTION "Audibly warns driver that seat belt needs buckling."
  FAILURE_CONSEQUENCE
      "Driver's attention not drawn to unbuckled seat belt
       Risk of injury."
    SEVERITY 3
    DETECTION 4
```

The same comments apply as to the earlier description of purpose. It will be seen that this means that if either of the effects is achieved correctly, the consequences of failure of the system, as shown in the resulting design analysis report, are less severe than if both fail. For example, the consequences of the horn sequence failing while the lamp comes on correctly (for example, if wire 'A' breaks), are those of `audible_unbuckled_warning` so the risk priority number (RPN) is likely to be lower than if both fail. The RPN might be raised by a high occurrence factor if the subsidiary function depends on a particularly unreliable component, of course.

This completes the functional description for the seat belt system's primary function, all that is required is to link the triggers and effects to the correct system properties. This will be discussed once the system's other functions (the temporary and permanent disabling functions) have been described. These do, of course, affect the primary function as they prevent its being achieved or more strictly its being triggered.

These two functions are arguably somewhat problematic as they do not really have a purpose of their own. Rather they negate the purpose of the primary function. Related to this is the fact that their effect is best regarded as an internal system state, the only effect is the absence of the otherwise expected effects of the `give_warning` function. The temporary disabling function is triggered by the buckling and immediate unbuckling of the driver's seat belt. This might be described like this.

```
FUNCTION temporary_disable
  ACHIEVES no_warning
  BY
    NOT give_warning.driver_unbuckled
    SEQ give_warning.driver_unbuckled [before 2000mS]
    TRIGGERS
    temp_disabled
```

Here the `driver_unbuckled` trigger from the main function has been reused, to ensure that both triggers map to the same system property. Also, there is no

temporal constraint on the buckling of the seat belt (`NOT driver_unbuckled` becoming true) but it has to be unbuckled again within two seconds. The associated description of purpose can be given as follows.

```
PURPOSE no_warning
  DESCRIPTION
      "Avoids distraction by belt warning system during manoeuvres."
  FAILURE_CONSEQUENCE "Driver distracted while manoeuvring.
                       Increased risk of collision."
    SEVERITY 5
    DETECTION 2
```

As this function's effect is an internal system property, it needs to be linked to the `give_warning` function to achieve any real purpose. This is done by adding its effect to the trigger of the `give_warning` function, like this.

```
FUNCTION give_warning
  ACHIEVES warn_unbuckled
  BY
    NOT temporary_disabled.temp_disable AND
    speed > threshold
    AND
    (driver_unbuckled
    OR
    (passenger_present AND passenger_unbuckled))
    TRIGGERS
    PIF show_warning_lamp
    AND
    PIF sound_chimer
```

One difficulty with this function is that the disabling of the system is described by the `temporary_disabled` function, but not how the temporary disabling of the system is ended. What is needed is another function that ends the disabling, like this.

```
FUNCTION end_temporary_disable
  ACHIEVES warning_enabled
  BY
    ignition_off
    TRIGGERS
    NOT temporary_disabled.temp_disabled
```

It is worth noting that these descriptions imply that the state of the ignition is irrelevant, except in this case, so the driver could, in principle, buckle up,

unbuckle within two seconds and only then start the engine and still have the seat belt system temporarily disabled. This arguably implies that the components concerned can change state even though they are not connected to the battery through the ignition switch, which is not the case in the schematic shown, so it will be the case that careful verification of the system's design will show that the `temporary_disable` function is not correctly implemented. It is felt that this is an appropriate simplification of the functional modelling here, though this might or might not be felt to be the case in actual practice.

The description of purpose associated with this function might be given as follows.

```
PURPOSE enable_warning
  DESCRIPTION "Ends disabling of seat belt warning system."
  FAILURE_CONSEQUENCE
      "System unable to warn driver of unbuckled seat belt."
    SEVERITY 5
    DETECTION 2
```

The need for a description of purpose for the `end_temorary_disable` function is arguably problematic, as illustrated by the weakness of this description of purpose. These functions do have a purpose so associating them with a description of that purpose is not unreasonable. The difficulty is more the fact that these functions' purposes really relate to the purpose of the `give_warning` function. There is also the difficulty of distinguishing between the failure of the `give_warning` function and the failure of the `enable_warning` function, both of which result in no warning being given when expected. One way of distinguishing these cases is to regard the `enable_warning` function as having failed if the `temp_disabled` effect is true.

The permanent disabling function is sufficiently similar to the temporary disabling one that there seems little to be gained by adding those examples. Indeed, there seems little reason why those functions should not share the one description of purpose. Therefore, in the interest of brevity, that function will not be described. It will also have a similar cancelling function, unless, of course, it is the case that permanent disabling is literally true and once triggered nothing should enable the seat belt warning system for the rest of the life of the car. This then completes the functional description of this system and the mapping between these descriptions and the system model can be considered.

11.1.4 Mapping between the functional and system models

It will be seen from the above that there is no reference to any actual aspect of the system in the functional descriptions. For example, while the functional description calls for a lamp to light up, it does not explicitly refer to the lamp in the schematic. That link is made after the completion of both the functional model and the system's structural model, the schematic in this case. This mapping between the two models was discussed in Section 5.6. This approach differs from that of (Price, 1998), and used in the tool developed in earlier work (*AutoSteve*) where the functional description made explicit use of the actual system properties. The approach taken here encourages the reuse of functional models for different systems that fulfil some purpose. For example, the system illustrated here makes use of electrical connections throughout, as shown by the schematic, but the functional description is equally applicable to a similar system that makes greater use of software and digital technology, for example, using network connections between the components. Another advantage of this approach is that it allows the functional and structural models to be constructed separately, so either model can be constructed first. This introduces the possibility that the functional model can be built early in the design process and used both as a representation of the functional requirements of the system and for verification of its evolving design. This is discussed more fully in Section 11.2.

The drawback is that there is a need for separate, explicit links between the functional descriptions' triggers and effects and the corresponding component states or properties in the system model. Note that in this example a qualitative simulator is used for the sake of simplicity, as described in Section 11.1.2, and that it is safe to regard the effectors (the lamp and chimer) as working if there is current flowing between their terminals. This is, arguably a fairly safe assumption in the case of the lamp as if it "blows" the filament breaks and no current will flow. It is less safe in the case of the chimer if it has failure modes that allow current to flow without it sounding.. If these assumption were not made, then behavioural descriptions of the components could be used as the basis for these mappings, and would also be required for simulation of the system, alongside the electrical simulator, as is described in (Snooke, 1999).

The triggers for the `give_warning` function mostly map to the states of switches within the system. Given that the components labelled "pass buckle" and "driver buckle" are switches within the seat belt fitting that are opened when the buckle is inserted, then the `driver_unbuckled` element in the trigger expression can be implemented either by the state of the switch, or even by indicating that the electrical resistance across the switch is zero.

```
‘‘driver buckle’’.position = unbuckled
IMPLEMENTS
give_warning.driver_unbuckled
```

The passenger buckle switch mapping is similar. A similar approach is used for the passenger seat occupancy detector (the “seat mat”) where if the seat is occupied, this closes a sprung push-to-make contact, so the mapping might look like this.

```
‘‘seat mat’’.position = occupied
IMPLEMENTS
give_warning.passenger_present
```

In both these cases, of course, a reusable switch model could (in principle) be used, provided care was taken that the “open” and “closed” switch positions were correctly mapped.

As noted earlier, the treatment of the speed sensor is simplified, so that there is no need to include the speed sensing system as part of this system model. As the seat belt warning system is merely concerned with whether or not the vehicle speed is over or under some threshold, we can get away with treating the speed sensor as a binary switch with two states, so that mapping might (rather simplistically) look like this.

```
‘‘speed sensor’’.state = fast IMPLEMENTS give_warning.speed > threshold
```

The remaining element of the `give_warning` trigger expression is the state of the disabled functions’ effects. These are discussed below.

That completes the trigger mappings for the `give_warning` function. The disabling functions mostly reuse these mappings, by specifying triggers such as `give_warning.driver_unbuckled`. The `temporary_disable` function is triggered using an expression that can reuse the mapping between the driver buckle component and the `give_warning` trigger. This avoids the need for redundant mappings, eliminating the danger that the two sets of mappings differ. The addition is the use of the ignition switch, specifically to cancel the temporary disabling function when switching off. As a car’s ignition switch typically has four positions (which can be labelled off, accessories, run, start) the correct subset of these positions has to be used in mapping between the system model and functional description.

```
ignition.position = off
OR
ignition.position = accessories
IMPLEMENTS end_temporary_disable.ignition_off
```

It will be seen that these mappings are quite straightforward. As discussed in Section 5.6, the mappings for the effects are more complex.

The effects for the `give_warning` are actually those associated with the two subsidiary functions, each of which has one effect. The effect `lamp_lit` of the `show_warning_lamp` subsidiary function can be mapped to the lamp, like this.

```
lamp.current = active IMPLEMENTS show_warning_lamp.lamp_lit
  UNEXPECTED_CONSEQUENCE "Dashboard lamp distracts driver"
    SEVERITY 3
    DETECTION 2
```

Note the inclusion of the consequences of unexpected achievement of the effect, which might be caused by the wire connecting the lamp to the GEM shorting to ground.

The effect of the `chimer_on` effect of the `audible_unbuckled_warning` function is a little more interesting. Here the mapping is between one element of the effect expression, like this.

```
chimer.current = active IMPLEMENTS audible_unbuckled_warning.chimer_on
  UNEXPECTED_CONSEQUENCE "Incessant chiming distracts driver"
    SEVERITY 4
    DETECTION 2
```

In both these cases, component states could be used in place of the values for current. This might well be required where a component is behaviourally more complex, and specifically where it has failure modes that mean that the overall system state (the presence of current) does not necessarily result in achievement of the required effect. This illustrates the advantage of associating the unexpected consequences of an effect with the mapping between system and effect rather than with the function. Here, it is most likely that the timing of the chimer is controlled by an ECU (the GEM) and this being the case, the wire connecting the chimer with the GEM shorting to ground will cause the chimer to sound incessantly. As this does not achieve the associated function (which depends on the ordering and timing of the effects) if the consequences of unexpected achievement were associated with the function, that failure would (apparently) have no effect.

It is also worth noting that here the value ACTIVE for the current has been used, consistently with the qualitative simulation described earlier, but these mappings could equally well be mapped to quantitative values, if a numerical simulator was to be used.

These are the only two effects associated with the `give_warning` function, but there is an effect associated with the `temporary_disable` function, which is then added to the trigger expression of `give_warning`. This might be described like this.

```
GEM.state.seat_belt_off IMPLEMENTS temporary_disable.temp_disabled
  UNEXPECTED_CONSEQUENCE "Driver not warned of unbuckled seat belt"
    SEVERITY 4
    DETECTION 7
```

This assumes some sort of state based behavioural model of the GEM. This aspect of the GEM was omitted from the discussion of the simulation for simplicity and as it was unnecessary for the illustration. This mapping is reused in both the `end_temporary_disable` function and in `give_warning`, as part of its trigger expression. The consequences of the effect being achieved unexpectedly are that the expected warning is not given when it should be. As these are identical to the consequences of failure of `give_warning`, a possible refinement might be to use these consequences (in the `warn_unbuckled` description of purpose) as the consequences of unexpected achievement of this effect. Similar mappings would also be required for the permanent disable function.

Having completed the mapping between the system model and its functional model, running simulations of the system in various states can show cases where the system's behaviour does not match that specified by the functional model. This can be used either to verify the design or for failure analysis.

The schematic of the seat belt warning system is believed to be correct, but if, say, wire 'C' was omitted from the schematic, then running a simulation will reveal that even when the speed is over the threshold and the driver is unbuckled, there will be no current through the lamp so it will not come on, even though it should, and the consequences of this error in the system design are known. If failure analysis is being carried out, then the system can be simulated with that wire broken (for example) and the same consequences will be found. A part of an FMEA report that could be automatically generated from simulation of this system, interpreted using the functional model described here, is shown in Table 11.1. It will be appreciated that the whole FMEA report will be considerably

Failure effect	Cause	Consequence	Sev	Det
Function give_warning failed because function sound_chimer failed because expected effect chimer_on was absent	wire A open circuit	Driver's attention not drawn to unbuckled seat belt. Risk of injury.	3	4
Function give_warning failed because function show_warning_lamp failed because expected efect lamp_lit was absent and function sound_chimer failed because expected effect chimer_on was absent	wire B open circuit	Driver not warned of unbuckled seat belt.Risk of injury.	4	7
Function show_warning_lamp achieved unexpectedly because unexpected effect lamp_lit was present	wire C short to ground	Dashboard lamp distracts driver	3	2

Table 11.1: Part of an FMEA report for the seat belt system example.

longer than this extract, typically an FMEA report will include several hundred rows. However, this short extract is sufficient to illustrate the main features of the approach, such as distinguishing between the complete ad partial failure of the main `give_warning` function. It has also been simplified by omitting any reference to the trigger expressions. This seems tolerable here as the failures shown will mean the associated effects and functions are never achieved correctly.

The descriptions of the functions that have failed or are unexpected and of the consequences are consistent with the rules for representation of failures introduced in Chapter 5 and summarised in Appendix A. This summary refers to the rule numbers as given in the appendix, as they are easier to find for reference. The top row of the extract shown notes that the function give_warning failed consistently with rule A.11 as are the failed subsidiary function and the missing effect (rule A.12). As only one of the subsidiary functions failed the consequences are those of that function, consistently with rule A.19. In the second row, both subsidiary functions have failed, so the consequences are those of the top level function, following rule A.18. In the third row, there is an unexpected effect leading to unexpected achievement of the lamp_lit function. As that function has no consequences of unexpected achievement, the consequences are those associated with the effect (strictly, its mapping to the system model), following rule A.15. It will be seen that this allows the report to distinguish between complete and partial failure of the top level give_warning function and also means that the consequences of any unexpected effect are included.

It is perhaps worth closing this section with a brief description of how this simulation and interpretation works, for one of the failures shown. Suppose the system model has been set with the input states (switch positions) as in section 11.1.2.

Also, as this is a failure analysis, the system is simulated with wire A open circuit. Running the simulator will show that the chimer never sounds, though the lamp will light. On completing this simulation (actually one of a sequence of simulations) then a comparison with the mappings and the functional model will reveal that the settings of the system mean the `give_warning` function is triggered, so both the purposive incomplete subfunctions are triggered, but that the `sound_chimer` function has failed (its trigger, inherited from `give_warning` is true but its effect false as the chimer never sounds) so this failure and the consequences of its associated description of purpose are included in the resulting report, as shown in the table. It will be appreciated how laborious a task this interpretation would be without the interpretation allowed by the functional description. It will be seen that a similar simulation and interpretation can be used for design verification by simulating the system with no failures and finding similar mismatches between expected results (as specified in the functional model) and the results of the simulation. This approach does, of course, follow the functional labelling approach described in (Price, 1998) with the difference that the specification of the trigger allows the use of the functional descriptions for design verification and the additional operators allow the description of the more complex behaviour associated with the chimer function.

The foregoing shows the usefulness of the approach for the intended purpose of automating design analysis of engineered systems. The relationship between functional models created using the Functional Interpretation Language and the simulation is discussed in more general terms in Section 11.4, following a discussion of possible other rôles of the language in the design of engineered systems.

11.2 Other possible rôles for the language

Having described a case study demonstrating the use of the Functional Interpretation Language for interpretation of simulation for design analysis in some detail, other case studies will be used to illustrate other uses of the language, and especially features of the language not fully explored in the preceding section. However, these case studies will not be described in such detail, in the interest of brevity.

Beside the use of language in automating design analysis (both design verification and failure analysis), as illustrated in the preceding section, there are two areas that might be considered as possible applications of the functional modelling the language enables. These are diagnosis and functional specification of a system design. It should be appreciated that these were not the motivations for

the research, so the usefulness of the language in these rôles has not been fully explored. This might be done in future, especially for the analysis of software based systems.

11.2.1 Diagnosis

As the Functional Interpretation Language is intended for use in design analysis, including failure analysis, it can be used for generation of a fault tree that can be used as the starting point for diagnosis. The use of functional interpretation of the simulation allows a functional description of the symptomatic system failure to be used, subject to correct matching of terms in relating the description of the symptom to the functional description of the failure in the fault tree chart.

The generation of a fault tree from FMEA was described in (Price *et al.*, 1996c) and the use of a fault tree in diagnosis for identifying candidate failures in (Price *et al.*, 1996b). These approaches are equally applicable to FMEA carried out using the Functional Interpretation Language.

One possibility that might be explored is that the more precise functional descriptions that the Functional Interpretation Language is capable of might be used to improve the candidate identification from a fault tree relative to the earlier approach. This will certainly be the case where a system includes functions whose trigger cannot be unambiguously derived from the simulation of the correctly working system (that is, there are dependent functions) as that need no longer be the basis of the failure analysis that forms the basis of the fault tree. This follows from the improvement the present approach offers for failure analysis if such systems, in that the functional description defines the trigger for all functions.

It is worth noting that diagnosis maps back from effect to cause of a fault in behaviour unlike the design analyses that the Functional Interpretation Language is intended to help automate, all of which map forward from cause to effect. In other words the FIL is intended more to help with forward chaining tasks. There is at least room for arguing that there is less need for the mapping to purpose in diagnosis as the symptoms are known and can be described in terms of an effect (in many cases) rather than the failure of a system to meet its purpose. However, this is not the case in all cases and there is a possible rôle for a functional language, such as the present one, in identifying candidate failures in such cases. Because the Functional Interpretation Language increases the capability of automatic design analysis and this can be applied to diagnosis as described in (Price *et al.*, 1996b), it follows that the language increases the capability of the approach in that paper.

11.2.2 Functional specification of system design

One advantage of the inclusion of the trigger and the use of labels for the trigger(s) and effect(s) associated with a function is that a functional description and so a system's functional model can be constructed independently of the system model itself. It can, therefore, be constructed before the system model, leading to the idea that the language can be used to specify the functional requirements of a proposed system in such a way that validating a design of the system against these requirements can be done automatically as part of the design process. This idea was introduced in Chapter 5, using the simple torch example, where it was shown that the functional model of the torch could be constructed to include a functional description that included the torch being lit while a button was pressed (for flashing the torch). This would imply the use of a push-to-make switch alongside the conventional slider switch.

Another aspect of this use of the language is possibility of using the language to specify what should happen in the event of failure of some system function. For example, in (Gunzert & Nägele, 1999), the development of a safety critical system in which a fault tolerant module is built from two "fail silent units" is described. The failure of a fail silent unit might be described using the Functional Interpretation Language as in the example below. This is a simplified description of the Brake By Wire Manager for an automotive braking system as described in that paper.

```
FUNCTION brake_requested
  ACHIEVES braking
  BY
    brake_request AND brake_force
    TRIGGERS
    signal_brake_pressure
```

This function describes (at an admittedly high level of abstraction) the normal behaviour of the brake manager, simply that it receives a request (`brake_request`) that indicates that braking is required and a request for a force of braking (`brake_force`), both derived from the position of the driver's brake pedal and in response sends a signal (`signal_brake_pressure`) to the four brake actuators (one for each wheel) requesting a certain degree of braking force to be applied. The system described in (Gunzert & Nägele, 1999) specifies the use of two such modules, the failure of either one can be tolerated, provided that it fails silently so no conflicting braking signals are sent to the actuators. The idea that the Brake By Wire Manager should fail silently can be specified by the Functional Interpretation Language, together with the condition that should trigger this function.

```
FUNCTION fail_silent
  ACHIEVES fault-tolerance
  BY
    (brake_request AND NOT brake_force)
    OR
    (brake_force AND NOT brake_request)
    TRIGGERS
    signal_silent
    AND
    close_down
```

Here if one of the input signals is detected and not the other, this implies a fault in the Brake Manager and it should stop working (having sent a signal to that effect), allowing its partner to govern braking alone, so the driver can continue until it is safe to stop. In other words, the braking system as whole carries on in a limp home mode.

In this case, the description has been simplified somewhat, but it does illustrate the idea that this failure mode behaviour can be specified as part of the functional description of the Brake Manager.

Another approach to the specification of a system failure mode behaviour is the use of dependent functions. For example in a motorised camera, pressing the shutter release causes a sequence of events to happen. As was shown in Section 10.4, these can simply be described as a sequence of effects (or subsidiary functions), like this.

```
FUNCTION take_photo
  ACHIEVES photograph_taken
  BY
    press_button
    TRIGGERS
    PIF expose_frame
    L-SEQ
    PIF wind_on
```

This covers three actions, the exposure (opening and closing of the shutter), the effects needed for `expose_frame`, the winding on of the film and the re-cocking of the shutter. However, this description does not define what should happen if one or other of these effects should fail. If dependent functions are used, the response to failure can be defined and so incorporated into the functional specification of the camera. One possible approach might be to reduce the `take_photo` function such that `expose_frame` is the effect, like this.

```
FUNCTION take_photo
  ACHIEVES photograph_taken
  BY
    press_button
    TRIGGERS
    expose_frame
```

As there is now but the one effect of `take_photo`, there is no need to treat `expose_frame` as a subsidiary function. The recharging function, `wind_on` is now triggered by the achievement of this function, and can itself usefully be divided into two functions, so that the failure mode behaviour is defined. The resulting functions might look like this.

```
FUNCTION wind_on
  ACHIEVES new_frame
  BY
    FUNCTION take_photo
    TRIGGERS
    film_wound_on
```

And this.

```
FUNCTION cock_shutter
  ACHIEVES shutter_ready
  BY
    FUNCTION wind_on
    TRIGGERS
    shutter_cocked
```

So each function is triggered by the achievement of the previous function in the sequence. It will be seen that now, if the frame is not exposed, the film should not be wound on, so preventing the waste of a frame. This example is arguably an oversimplification, as `take_photo` might fail if, say, the timing of the shutter opening and closing was incorrect resulting in the frame being badly exposed. The other example is safer as if the film winding on fails, the shutter should not be cocked. This behaviour eliminates the danger of the frame being inadvertently double exposed. This demonstrates the use of the Functional Interpretation Language to specify aspects of the failure mode behaviour of a proposed system as part of its functional specification. This increases its usefulness for the functional specification of a system. Thus example shows how the language might be used (if somewhat informally) to refine the system's functional specification. The specification of the motorised camera might start with the simple description, where the

three associated functions share a trigger, and this description might be refined as the drawbacks of this simple functional specification become clear. This might happen in response to examining the results of simulation as part of the design analysis, as the functional model is capable of this automated evaluation. For example, a simulation might show that the film does wind on despite the failure of the shutter to open but that this winding on is not itself seen as a failure, that is it has no consequences in the design analysis report. This could prompt the designer to create a more refined functional description that does highlight this failure, such as the more complex version given above.

The idea that the language is used in specifying the functionality of a system, as discussed in this section, allows the functional modelling and the simulation and so the design analysis to be integrated more closely in the design process. This encourages design analysis to be carried out early and often in the system's design process, when it is most beneficial.

A rather more speculative area related to this use of the Functional Interpretation Language is whether a system's functional specification constructed using the language can be combined with knowledge of the behaviour of candidate components to automatically generate a candidate system design, in an approach not unlike that of (Iwasaki *et al.*, 1993). This use of the language does differ from the approach in that paper, component functions are not used, being replaced by knowledge of component behavioural models, of the kind used in simulation for design analysis. For this to work the functional description of a system would need to be decomposed rather more than the "black box" description used in this thesis. One objection to the Functional Interpretation Language for this task is that the lower level functional description might be unwieldy as description of purpose might be felt to be redundant. One possible approach might be to decompose a system function into subsidiary operational incomplete functions, avoiding any need to construct superfluous descriptions of purpose. Because these subsidiary functions can use the effect of some other function as its trigger it is not impossible that a complete functional model could be constructed in this way and its appropriateness evaluated against the trigger and effect of the original top level function. This evaluation could be done before the functional model is reified as a candidate design, simply by comparing the truth of the conditions (trigger and effect) of the main function with the conditions resulting from the causal model constructed by combining subsidiary functions in this way. It should be stressed that this has not been tried and while the Functional Interpretation language might be insufficient to automate this process, it does seem possible to use it to help construct and evaluate candidate designs. How these functional models are

then mapped to a structural model of the system remains as an area for future work, as does developing this possible use of the language itself. One possible field in which this might be done is software design, if only because the knowledge of component behaviour might be available. Components in this case will, of course, be software components, such as object oriented classes and methods.

11.2.3 Explanation generation in education and training

One area where model based simulation has been used is in training and education. Examples include the use of the General Architecture for Reasoning about Physics (GARP) (Bredeweg, 1992) in learning. Developments for this application have included a graphical representation of the simulation models (Bouwer & Bredeweg, 2001) and an interactive tool to support the construction of the simulations (Machado & Bredeweg, 2001). In addition, *CyclePad*, a simulation tool for thermodynamic systems based on a process centred ontology (Forbus, 1997) was intended for use in education. It has been used in the instruction of engineers in the field of thermodynamics, (Baher, 1999) and its use was evaluated in (Tuttle & Wu, 2001).

The rôle of interpretation of simulation might well be rather different in this field, as compared to design analysis. For example, students might be encouraged to predict the result of a simulation and compare their results with the result of the model based simulation. While this places greater emphasis on the simulation itself, one can see a need for some means of providing some interpretive explanation of any differences between these results. These explanations will frequently be in terms of the relationships between the causes of the differences between the predicted and actual simulations rather than the effects of these differences, but it is not impossible that there are cases where the Functional Interpretation Language might prove useful. One possibility that springs to mind is in explaining the effects of errors in student exercises in programming. This remains a possible area for future work, however, as this use of the language has not yet been investigated.

11.3 Functional description of software

One of the motivations for the development of the Functional Interpretation Language is the desire to automate design analysis of systems that incorporate software based components. This was the aim of the SoftFMEA project which was the source of funding for much of the research described in this thesis. It will

be appreciated that the use of software introduces the greater complexity of behaviour on which a system's functionality can depend, and which the language was developed to describe.

While software examples have been used in this thesis, the fact that the description of software based systems was an important part of the motivation suggests that a brief discussion of the use of the language for describing software is worth including. This section will discuss questions that arise from the use of the Functional Interpretation Language for describing software and software based systems rather than include a full case study. This is partly because there are such questions that need some discussion and partly because any case study of realistic scale will be of sufficient complexity to make the necessary explanation excessively protracted.

The first question that arises is the one of whether the approach to modelling of software based systems for design analysis is of interest. There has been interest in design analysis (particularly FMEA) of software and embedded systems (Maier, 1995; Bowles & Wan, 2001). With the increasing use of programmed components (such as ECUs) in areas such as the automotive sector the safety analysis of such systems is of increasing importance, especially with the introduction of "drive by wire" systems with no mechanical backup, as noted in (Papadopoulos *et al.*, 2001). That paper describes a "semi-automatic" approach to this safety analysis, based on constructing fault trees of the system, which process is repeated at finer granularities as the design process continues. The lack of, and need for, mature techniques for manual software FMEA is noted by (Goddard, 2000). There is a description of the early research into an approach to modelling software for automated safety analysis in (Snooke, 2004). That approach traces the paths of possible errors arising in a program's variables and so traces the effects of such errors. There have been encouraging initial results with the use of the Functional Interpretation Language for interpretation of the result of this analysis of software.

Using the Functional Interpretation Language to build functional models of the system as a way of specifying the system's intended functionality is, of course, equally applicable to software system as to any other system, so the possibility exists of building functional models of an intended software system early in the design process and being able to simulate the behaviour of the system so that the design can be automatically verified against this functional specification. This approach is not a substitute for a conventional formal method, because the correctness of the system described can only be demonstrated by simulation (so depends on the correctness of the system model) as opposed to being mathematically proved. In the case of software, of course, it is feasible for the source code to be used in the

system model, which means that model's correctness depends on the compiler. It does share with formal methods the idea that the functional requirements can be expressed in a such a way that the system can be verified against them and this will be done early in the design process.

While the `CYCLE` operator gives the Functional Interpretation Language a way of describing iteration in a program, the lack of any control operator might be seen as preventing its use in modelling the function of software. However, this is arguably not the case, as any control branching of a program that results in a different output (or effect) will simply result in the achievement of some other system function. This does mean that the condition that triggers the branch program needs representing in the functional description. For example, consider an Automatic Teller Machine (ATM) that will refuse a request for a withdrawal if the account has insufficient funds available. A possible pseudocode listing of the relevant fragment might look like this.

```
void checkBalance (int request)
  if (balance < request) {
    refuseWithdrawal ( );
  } else {
    dispenseCash (request);
  }
```

This does, of course, lead to two alternative responses to the original trigger (which can be taken to be the user entering the amount to be withdrawn). As the two responses fulfil different purposes, they are modelled as different functions in the Functional Interpretation Language, as shown here.

```
FUNCTION make_withdrawal
  ACHIEVES cash_dispensed
  BY
    request_withdrawal
    AND
    (request_amount <= account_balance)
    TRIGGERS
    dispense_cash
```

Here, of course, the variable `request` in the pseudocode implements the functional description's element `request_amount` and the `balance` variable implements `account_balance`. The purpose of this function is, of course, to dispense the requested amount of cash. The other branch of the ATM pseudocode results in the triggering of a different system function, the refusal of the request.

```
FUNCTION refuse_withdrawal
  ACHIEVES overdraft_avoided
  BY
    request_withdrawal
    AND
    (request_amount > account_balance)
    TRIGGERS
    request_refused
```

The mappings are shared with the `make_withdrawal` function. The purpose here is, of course, to protect the account from becoming inadvertently overdrawn and the effect is likely to be the display of some appropriate message on the ATM's screen.

Where a program's control path does not affect the actual effect or the purpose of a program or program fragment it can safely be ignored in the functional model of the system. Examples are not easy to come up with as these cases are relatively unusual. One possible case might be that it is specified that a program should not be closed with any unsaved work open, and while this leads to two distinct behaviours (if there is any unsaved work it must be saved, if not no action is necessary), functionally the end result is identical and so the difference between these behaviours could be ignored. It is appreciated that this is a somewhat contrived example which does, perhaps, show how unusual such cases are.

While the functional descriptions above are more elaborate than the code they describe, it is worth pointing out that they do incorporate an abstract view of parts of the software not listed, as the system functions described also depend on the correct behaviour of the `refuseWithdrawal` and `dispenseCash` code and indeed the hardware that actually dispenses the cash, so the system these functional descriptions describe is more elaborate than the illustration suggests. This example illustrates what might well be the most likely use of the Functional Interpretation Language with regard to the modelling of software, which is the modelling of embedded systems where at the functional level there is little need to distinguish between the hardware and software components of the system. Indeed the fact that the functional descriptions are independent of the domain means that the same functional description can be used whether a given function is implemented in hardware or software.

Having considered different uses of the Functional Interpretation Language and seen that its use for design analysis as well as for functional specification depends on the availability of a simulation engine to generate a model of the system's behaviour, it is time to examine the relationship between functional models built using the Functional Interpretation Language and the simulation engine.

11.4 Relationship with the simulation engine

As was suggested earlier, the automation of design analysis depends on both the ability to simulate a system, to obtain a description of the system's behaviour and the ability to interpret that behaviour in terms of the system's purpose to allow the automatic generation of a design analysis report. While the purpose of the Functional Interpretation Language is this interpretation side of the design analysis task, it is necessary to at least consider the implications of the use of this language on the requirements of the simulation engine (or engines) used in a design analysis tool. As the subject of this thesis is the interpretation language, rather than simulation, this discussion will be confined to requirements of the simulation. The use of the Functional Interpretation Language should not restrict the choice of simulator, provided the chosen simulator provides output that can be mapped to the appropriate parts of the functional model and it meets the requirements discussed in this section. While an electrical simulator was used in the worked example earlier there is no reason why that should be the case. The seat belt warning system could have had more of its control behaviour implemented in software (using a data bus for communicating between the ECUs, perhaps) and while that would lead to a change in the approach for simulation, there would be no need to change the functional description. One could envisage a design analysis system that used the FIL for interpretation and allowed the use of alternative simulators for electrical, hydraulic and software based subsystems, and any others found necessary.

This follows from the functional labelling approach in (Price, 1998) and a brief summary of that system as implemented in the commercial design tool (*AutoSteve*) will help to clarify the requirements to be discussed. This approach uses an electrical simulator combined with behavioural descriptions of the components, as described in (Snooke, 1999). To run a simulation, the inputs to the system (such as switch positions) are set and the simulation started. The simulation will then run until the system settles into a steady state whereupon the states or activity of the effector components can be mapped against the functional labels to establish which system functions are achieved in this system state. For example, if after the simulation of the current behaviour of a car lighting system is complete and the system is in a steady state there is current in the headlamp dipped beam filaments, the `dipped_beam` function is achieved.

This introduces one immediate difference as using the Functional Interpretation Language the function will only be achieved if the function's triggers (which map to the input components) are also true. In other words, the function will only be

achieved if the switches are set correctly for that function. This does not place any new demands on the simulation engine as the switch positions were known at the start of the simulation.

There are a few areas where the relationship between the Functional Interpretation Language and the simulation needs discussion. These arise largely from the novel features of the language. They will be discussed in turn.

11.4.1 Sequential behaviour and temporal constraints

It will be appreciated that the approach described above is appropriate where a function depends on a persistent output or state of the system, but cannot be used with functions that depend on intermittent or sequential outputs, either because the system will never reach a steady state until some other input causes the intermittent behaviour to stop or because the earlier outputs in a sequence are over before the steady state is reached. These are, of course, the types of function that can be described using the `SEQ`, `L-SEQ` and `CYCLE` operators. Therefore the ability of the Functional Interpretation Language to describe functions that depend on such behaviour introduces these two new requirements on the associated simulation engine. In practice the simulation engine has to recognise that the simulation has entered a cycle regardless of the interpretation, simply because if it fails to do so, the simulation might run indefinitely. There seems to be no intrinsic difficulty in having the simulation engine returning a sequential description of the states the system entered during the simulation. The example simulator output in Section 11.1.2 does this. The only difficulty is ensuring that such a sequential description of the system's behaviour covers all the significant states. Obviously, if every state change is listed, this will be done, and the interpretation side of the design analysis tool can ignore those state changes that do not affect the state or behaviour of any components that are mapped to the functional model. For example, the momentary state where a relay has current flowing through it, but the switch has not yet moved in response can be ignored as the effector components driven by the current through the relay's switch will not have changed state.

This requirement of the simulation engine can be further refined by requiring the modelling of timing of these changes of state, so that the timing of achievement of a function can be described, allowing the interpretation side of the design analysis tool to identify situations where a function is achieved in an untimely manner, as discussed in Chapter 9. This does entail the incorporation of timing information in behavioural models, of course, which increases the overhead in building the necessary models. This is quite applicable to models based on state charts,

as described in (Snooke, 1999). This approach is also appropriate for software based behaviours. Another aspect of timing in a system model is where timing constraints are a part of the trigger of a system function, as in the temporary disabling function of the seat belt warning system described earlier in this chapter. This does raise problematic requirements for the simulation engine, as an input event will have to be added to the simulation at the appropriate point while the simulation is running following the previous input. The assumption that inputs to the system are independent and instantaneous can no longer be used. One possible (though perhaps not feasible) approach is to run the simulation in real time (perhaps scaled if necessary) so the new inputs can be made when needed. Another possible (if complex) approach is to allow a programmed scenario of inputs (that would otherwise drive distinct simulation steps) to include temporal information. Then, given that the simulation can keep track of time, the timed input can be made programmatically at the right point in the simulation.

11.4.2 Simulation through the design lifecycle

Because the Functional Interpretation Language allows the possibility of building the functional model of a system early in the design process, as part of requirements definition, this raises the possibility of using the functional model to interpret a sequence of different simulations, of finer granularity, as the design process progresses. This is not dissimilar to the generation of fault trees of finer granularity proposed in (Papadopoulos *et al.*, 2001). An approach to analysis using finer grained as more precise models become available is described in (Price *et al.*, 2003). Here as the electrical system design is refined, three valued qualitative circuit modelling is superseded by multiple valued qualitative and finally by numerical circuit models.

The functional descriptions are, of course, equally applicable to any of these system models. All that is required is a refinement of the mappings between the system models and the effects in the functional description. For example, a system function might depend on some lamp being lit, and the lamp might be regarded as being lit if the current through it is active in the three valued model, medium in the multiple valued qualitative model and with a current of, say, between 500 and 700 milliamps in the numerical model. The effect's label need not change, so neither need the functional description itself; only the mapping between it and the system model changes. The functions to which the lamp being lit contribute do not change at all, of course, as the purpose will not. Even this change to the mappings between the system and functional descriptions might not be needed if

the effector components are described in terms of state as the state description will themselves change with the increasing granularity of the model, but the same (or at least corresponding) states will be associated with a function's effect. This places all the changes required through the design process on the simulation side of the tool, with the advantage that as they are associated with reusable component behavioural models. These alternative models can be retrieved from a library of component models.

11.4.3 Dependent functions

The idea that a system function might depend on some other function (as discussed in Chapter 10) has relatively little effect on the simulation side of a design analysis tool. One point that does arise, though, is that a where a function is triggered by the achievement or otherwise of some other functions, it will be triggered as a result of a change in system state during the simulation rather than directly by the inputs from outside the system. There seems no reason why this should cause any particular problems, but it is the case that the sequential description of the system behaviour required for functions having intermittent or sequential effects will also need to be checked by the interpretation side of the design analysis tool for system states that trigger some dependent function. The idea that a dependent function might affect the seriousness of the main functions, such as a telltale light failing and making the failure of the main function less detectable is a purely interpretive point and has no effect on the relationship between the simulation and the functional model.

Having discussed the aspects of the Functional Interpretation Language that might have most affect on the relationship with the simulation engine, there are one or two minor aspects that can be swiftly discussed.

As the functional descriptions are generally independent of the domain of the system, there should be no difficulty relating the functional descriptions to systems whose analysis depends on a mixture of domain based simulators, such as there being domain based simulators for the electrical and water flow in a washing machine.

Because the triggers and effects of a function are essentially binary in nature, being either true or false, there is a need for these to be related to states of components that are not binary. This might be because the output (say) is modelled numerically, in which case an acceptable range of values can be associated with correct achievement of the effect, as described above, or because the effect depends on a component with several qualitatively distinct states (such as slow or fast of

windscreen wipers). This presents no difficulty as again the mapping between the functional description and the system model will use the correct one of these possible states and logical `NOT` can be used to simplify functional descriptions where necessary.

All the foregoing has been concerned with the relation between the Functional Interpretation Language and a system model and simulator built using a component centred ontology. This is, arguably, a natural ontology for the design analysis of engineered systems. It is also the case that other functional modelling approaches tend to be associated with that ontology. This is clearly the case where the functional language is used to define the function of components, such as in (Hawkins & Woollons, 1998).

However, while the present work has not investigated the possibility of using the functional language for interpretation of the effects of a process based simulation, there seems to be little difficulty in doing so. A function is modelled in terms of the triggers and effects on a system's intended behaviour, associated with specific components (such as the headlamps in the example above). It should be possible to associate system functions with specific processes rather than specific components. Indeed, the functional model might be identical. For example, a boiler needs a function that prevents its pressure climbing above a safe level, like this.

```
FUNCTION reduce_pressure
  ACHIEVES prevent_explosion
  BY
    pressure > working_pressure
    TRIGGERS
    release_steam
```

There seems to be no difficulty in mapping the effect either to the state of a component (the safety valve) or to the triggering of some process, the escape of steam.

The discussions above, regarding the relationship between the Functional Interpretation Language and the simulation engine suggest that a design analysis tool has two distinct (but mutually dependent) aspects, so there is a simulation engine (or more than one domain based one) forming the simulation side of the tool and the Functional Interpretation Language forms the basis for the interpretation side of the design analysis tool. In principle, the domain independent nature of the Functional Interpretation Language allows different simulators to be used in conjunction with a functional model, simply by changing the mappings between the system model and the functional model, so it is not impossible, say, that the same functional model could be used for a gas and electric hob.

While this is not concerned with the relationship between the interpretation and simulation, it seems worth concluding this section with a brief consideration of how the arrangement of functional descriptions, so that the functions can share effects and mappings, might be implemented. This is especially important where there are dependent functions as this increases the number of relationships between functions, and so cases where ne functional description will refer to another. The approach taken in this thesis is to use a qualified name for the function or element that is being referred to, as was illustrated in the seat belt warning case study earlier in this chapter. For example, the `temporary_disable` function has as its trigger the trigger of the seat belt system's main function, `give_warning`. This was shown by qualifying the label of the trigger with the name of the function to which it belonged, `give_warning.driver_unbuckled`. One approach to implementing these relationships is to keep the functional descriptions in separate files, having the arrangement of files mirror the arrangement of functions and elements. A group of functions might be associated with a system, say, so that the system is mirrored by a directory with the same name, and functions can therefore be identified by the system, as well as elements (such as trigger and effect) by the function. This is very similar to the way the package structure in a Java application's source tree matches the file structure itself.

Chapter 12

Conclusion and future work

The use of the Functional Interpretation Language for interpretation of simulation of engineered systems, as has already been stated, builds on the Functional Labelling approach of (Price, 1998) and by the use of that approach in *AutoSteve*, the model based design analysis tool developed in previous work in Aberystwyth. That work demonstrates the effectiveness of the approach on which the present work builds. Therefore the conclusion of the present work must be in terms of how effectively the language described herein addresses the limitations of the earlier work, by increasing the range of systems and tasks for which the language can be used, and more specifically by how well the Functional Interpretation Language meets the sets of requirements stated in the Introduction and in Chapter 5.

This chapter will discuss the language in terms of those requirements and in relation to the research question, *can we devise a language for interpretation of behavioural simulation of engineered systems (of arbitrary complexity) in terms of the systems' purpose?*, and motivation presented in the introduction before a discussion of possible future work. This discussion of future work also serves to highlight other areas in which the Functional Interpretation Language might prove to be of use, though its use in these areas has not been explored as part of the present research.

12.1 How the Functional Interpretation Language meets its requirements

This section discusses how well the functional interpretation language meets the requirements set out in Chapter 5 and then goes on to discuss how well it meets

the requirements for functional modelling of systems of the kinds enumerated in the Introduction.

There are five requirements of a functional modelling language set out in Chapter 5, each will be discussed in turn.

Capable of recognising whether the system's purpose is being fulfilled. This feature of the Functional Interpretation Language is common with the Functional Labelling approach of (Price, 1998) in which the achievement of a function is associated with the presence or otherwise of some expected effect. The present language can be argued to refine this approach by the inclusion of the trigger (which was added to the functional labelling approach for design verification) leading to the abstraction of a system's behaviour into one of four states expressed largely (though not entirely) in terms of the fulfilling of some purpose of the system. The use of the recogniser (based on the truth of the trigger and effect expressions) allows the language to recognise when a system's purpose is being achieved.

Applicable to static or dynamic functions. This is, arguably, the weakest aspect of the Functional Interpretation Language. The nature of the recogniser of achievement of a function, depending as it does on a trigger and effect, can hardly be said to lend itself naturally to the description of a static function, though an approach to modelling such functions was described in Chapter 5. As the primary aim of the language is interpretation of simulation (where it is reasonable to expect some dynamic change to the system), this weakness is, perhaps, understandable. However, the approach to describing static functions in Chapter 5 is a sound approach to the problem so the langauge does fulfil this requirement. What it is incapable of describing usefully are functions whose effect is incapable of simulation, such as a subjective case like the aesthetic function of a placing a vase of flowers in a room. The language could be used to describe such functions, but their achievement cannot be recognised, as it cannot be modelled in the simulation.

Applicable to physical and abstract objects. The most significant example of abstract systems we might want to analyse are software based ones, and the use of the Functional Interpretation Language for the design analysis of such systems was discussed in Section 11.3. The language can be used for the description of abstract systems, with the proviso that such systems must (assuming the language is used for its intended purpose of the interpretation of simulation) be capable of being simulated in terms that are applicable to the language. This implies

the presence of triggering inputs and outputs to and from the system that can be associated with the triggers and effects of the system's function. It seems reasonable to suggest that this is an appropriate model for analysis of software based systems (such as programmable electronic control units). The language could be used to describe such abstract systems as elements in a work plan (such as specifying the triggering inputs and effects of individual work packages) but this has not been explored and the usefulness of the language in this area has not been established.

Independent of the system. This is a significant difference between the Functional Interpretation language as presented here and the functional labelling approach from which it is derived. In the functional labelling approach in (Price, 1998), the functional description is attached to an known property of a component of the system to be analysed. Therefore, the functional model could not be built independently of the system with which it was associated. The use of labels for the trigger and effect of a function, that are then mapped to the appropriate system properties, does allow a system's functional model to be constructed independently of the system. One advantage of this is the possibility it introduces of building a functional model of an intended system before the system itself is designed, allowing the functional model to specify the functionality that the system is to embody and allowing the system to be tested (in simulation) against this specified behaviour. The functional model can therefore be regarded as representing the system's functional requirements. This is arguably the most significant improvement over the earlier work.

Capable of specifying expected behaviour to arbitrary level of precision. While the functional hierarchy used in the functional labelling approach, as described in (Snooke & Price, 1998), allows a system function to be described with precision, the use of both triggers and labels for functions and effects encourages a greater degree of precision in describing a system function. This was discussed in Chapter 5. The simple torch example used in that discussion serves to illustrate the advantage of the present approach, illustrated by the example of the use of the language in specifying the need for a trigger for flashing the torch. It is also suggested that the use of labels to allow decomposition of functions in terms of effects as well as in terms of subsidiary functions is valuable in constructing a precise functional description.

The list of requirements above includes two of the three requirements of a functional modelling language in (Chandrasekaran & Josephson, 1996) and it seems

worth at least mentioning the remaining one, that the language be applicable to natural and man made systems. As was argued in Chapter 5 this was felt not to be important in a language intended for design analysis of man made systems and there is arguably a difficulty over considering the behaviour of some natural systems in terms of purpose. However, it does seem possible that, given that a system can be described in terms of fulfilling a purpose, the Functional Interpretation Language might be useful for modelling natural systems. For example, it would be quite feasible to use the language to describe the function of an organ (such as the heart), but given the problems of any notion of purpose introduces, it is perhaps inappropriate for describing natural ecosystems. It could conceivably be used to describe biological systems resulting from human use of land, where the notion of purpose is less difficult. This has not been explored, however.

As was stated in the introduction to this thesis, the original motivation for the development of the Functional Interpretation Language was to increase the range of systems for which the functional labelling approach to model based design analysis can be applied. The modelling of the four classes of system behaviour concerned was discussed in Chapters 6 to 10 and how these features of the language might be used in practice was discussed in Chapter 11. The importance of this work in motivation of the research herein suggests that it is worth a brief recap of these features and their use in this concluding chapter. There are four classes of system behaviour that it was felt the language should be capable of describing.

Systems where a function might be partially achieved. Unlike the approach to hierarchical functional modelling described in (Snooke & Price, 1998), the Functional Interpretation Language draws a distinction between a function's effects and subsidiary functions. This allows the language to differentiate cases where the presence of one of two (or more) effects can be regarded as partial achievement of a function from cases where the function depends on all effects being present, as discussed in Chapter 6, where these cases were illustrated using small examples. The flexibility of this use of subsidiary function is increased by the introduction of "incomplete functions" so that common parts of a functional hierarchy need no repetition. This was illustrated in Chapter 7. Therefore the language allows cases where a function is partially achieved to be distinguished, where this distinction is appropriate. The examples used in those earlier chapters (the headlamp system against the warning system) illustrate this distinction. This was further explored in the full case study in Chapter 11 in which one of the visual and audible outputs occurring despite the loss of the other could be regarded as mitigating the loss of the warning function.

Systems whose functionality depends on intermittent or sequential effects. This is, perhaps, the most apparent addition to previous functional description languages. The addition of sequential operators (similar to the future operators in temporal logic) introduces the possibility of functional description of a range of systems that could not be so described previously. This was discussed in Chapter 8, which included examples of such systems and the use of these operators was also explored in the case study in Chapter 11. As has been noted, the fact that the system described in that chapter could not be modelled with the earlier language was a trigger for the present research. The use of the sequential operators allows more precise description of the temporally complex behaviour to be expected of software based systems and the increasing use of such systems embedded in many products enables a valuable increase in the range of systems the language can describe, relative to earlier functional description languages.

Systems where there is a danger of the required effects being achieved in an untimely manner. This feature of the Functional Interpretation Language clearly relates to the sequential outputs summarised above. The use of temporal constraints to specify the timing of the sequence of the chimer in the seat belt warning system was illustrated in Chapter 11. The distinction between the untimely achievement of a function (as opposed to its unexpected achievement) and its failure was discussed and illustrated in Chapter 9. The possibility of such untimely achievement of a function is increased by the increasing interrelationships between system functions. One example of this is where functions depend on competition for access to a CSMA-CD network (such as CANbus) so that network messages are delayed by the network being occupied by the messages associated with some other (arguably higher priority) function. The use of these temporal constraints allows such cases of late achievement of a function to be recognised and differentiated from failure of the function.

Systems with subsidiary functions whose achievement depends on the state of some other system function. The inclusion of the trigger as the precondition of a function introduces the possibility of the post-conditions (effects) of a function being used as the precondition (trigger) of some other function, so that the idea that one function depends on the state of another can readily be described. This allows the description of the different categories of function that depend on some other function that were discussed in Chapter 10. The set of telltale and warning functions described therein is complete (given their dependence on inputs and outputs rather than internal behaviour), but the fault tolerant func-

tions set cannot be so described as the nature of that set depends on the behaviour associated with the fault tolerant function. There is also the recharging function that was illustrated by the camera example in Chapter 11. A functional language that does not include the trigger cannot describe such functional dependencies. An effect of this limitation is that such a functional language cannot describe the failure of such a function. The failure of a telltale lamp, for example, cannot be described, because the language fails to unambiguously define that function's trigger. The inclusion of the trigger in the Functional Interpretation Language eliminates this shortcoming.

These summaries have been kept brief, as these features and uses of the language were discussed in the chapters indicated. It is felt that any further description of this material in this conclusion is somewhat repetitious and therefore superfluous.

The fact that the Functional Interpretation Language can be used to describe the functions of systems with the above characteristics demonstrates that the language increases the range of system that can be modelled for design analysis, relative to the earlier functional labelling approach. In addition, the incorporation of a function's trigger and the use of labels for the function's trigger and effect, so that the functional model can be built independently of the system model, increase the range of tasks for which the language is applicable, as well as encouraging the closer integration of functional modelling and simulation into the design process.

Because the language depends upon a simulation, rather than being a self contained model of the system, it cannot be proved that the language can be applied to systems of arbitrary complexity. However, the language's consistency and the inclusion of temporal operators suggest that the limitation on complexity is more likely to be set by the complexity of the models than any inherent shortcomings with the language. The examples used in this thesis demonstrate that the Functional Interpretation Language can be used to describe the functions of systems whose behaviour was beyond the capabilities of the earlier language. This enables the simulation of such systems to be interpreted in terms of the system's purpose and so allows the automatic generation of design analysis reports both for design verification and for failure analysis of a wide variety of engineered systems, most particularly for software based systems.

12.2 Future work

Some possible areas for future work based on the present research were outlined in earlier chapters. It seems worth reprising these to summarise different ways in

which this research might be carried forward.

One difficulty with the present state of this research is the additional requirements on the simulation side of a model based design analysis tool. The Functional Interpretation language would support the design analysis of systems whose behaviour is beyond the capabilities of the implementation of an associated simulation engine. As was suggested in the section on the relationship with the simulation engine, Section 11.4, there is no intrinsic difficulty with building a simulation engine that fulfils the requirements that the Functional Interpretation Language places on that side of the design analysis tool. Indeed the approach to simulation used in *AutoSteve*, mixing a domain based simulator with state based component simulation, offers the possibility of a generating a sufficiently sophisticated description of a system's behaviour, but there is a need for an implementation of such a simulation engine that fulfils this potential.

An interesting possibility for simulation (or behavioural modelling) that might be used in conjunction with the Functional Interpretation Language is in the field of software design analysis. The approach to software modelling, based on tracing the possible paths of errors in variables that was introduced in (Snooke, 2004) seems to be a promising approach to model based design analysis of software systems. The basis for the system model (which forms the basis of the behavioural analysis) is, of course, the source code. There have been some encouraging early results using the Functional Interpretation Language to interpret this software modelling for generating FMEA of software. The modelling of software (especially of embedded software) looks a promising application area for this language, with its capabilities of describing the sort of complex behaviour that software can introduce into a system.

A possibly related area of future investigation is the use of the langauge in interpretation for explanation generation in education and training. As was briefly discussed in Section 11.2.3, this might be found useful in assessing and explaining the effects of errors in student programming exercises. This introduces a possible rôle for the language in education, by using it to enrich explanation of the workings of some system. There is possible scope for investigation of this use of the language in explanation generation relating to physical (and biological) systems, and possibly in interpretation of the results of exercises in modelling social and economic systems.

On a more speculative level, the possibility of using the Functional Interpretation Language for supporting the design process itself, although not a primary motivation of its development, does look an interesting area. One particular area that looks worth investigation is the possibility of using the language as a means

of specifying the functional requirements of a design, against which a candidate system design can be analysed by simulation, as the possibility exists of automatically comparing the system's behaviour (as simulated) with the functional requirements. This does have the advantage of incorporating the simulation of the system more closely in the design process, encouraging the early and frequent carrying out of the sort of safety analyses the functional language is intended to help automate.

It is worth noting that all the examples in this thesis have been modelled using a component centred ontology, indeed most approaches to functional description, at least for design analysis, seem to use that ontology. In the case of the present approach, as in the earlier functional labelling approach of (Price, 1998), system functions are associated with the inputs and outputs, or states, of specific components. This is an intuitive approach to use for the analysis of system design, in that it models the system as an assembly of components. However, one possible area of interest for future work might be to evaluate the use of the Functional Interpretation Language in relation to process centred simulation. While this is speculative at this stage, there do not seem to be any intrinsic difficulties with associating the trigger or effect of a system function with the state of a process rather than that of a component. One difference is that while physical components are invariably present in the system, even if they are idle, processes might appear and disappear during the course of a simulation. It remains to be seen whether this would lead to problems with associating a function with a process. One other question that arises is what properties a process has that can be associated with the trigger and effect conditions of a function.

A final possibility for future research, arguably the only one that involves altering the functional language itself is the investigation of whether the functional language can be used for supporting the design process by enabling the functional refinement of a system design. As the Functional Interpretation Language is not intended for the description of component function, it does not look ideally suited for this task, especially as the language does not represent the connection between components. While this is not an intended rôle of the language, the possibility that the langauge might be extended to allow its use for this task is a possible area for investigation.

To summarise, while the Functional Interpretation Language cannot be proved to support the functional description of systems of arbitrary behavioural complexity for automation of design analysis, it does succeed in increasing both the range of systems and range of design analysis tasks for which the approach is available.

Appendix A

Formal description of the Functional Interpretation Language

This appendix contains a formal description of the Functional Interpretation Language. It will therefore act as a key to the examples in the text of the thesis as well as providing a summary of the language's logical basis. There is little explanation herein, as that is in the text of the thesis.

A.1 Elements of the language

Let f be some device function. It is associated with a trigger t and an effect e where t and e are Boolean expressions. The trigger and effect expressions are mapped to appropriate properties (such as the states of relevant components) in the system's structural and behavioural model. In addition, f is associated with a description of purpose p_f that in turn includes a description of consequences of failure to achieve the function, c_f and may optionally incorporate a description of consequences of unexpected achievement of the function's effect u_f. It is these descriptions of consequences that will appear in an automatically generated design analysis report. This is annotated as R (for report) so that where a design analysis report will include the consequences of failure of f, this is indicated by $\mathrm{R}(c_f)$. The term "includes" is used to indicate that a function is associated with an optional element.

Each element n in an effect expression is associated with a description of consequences of unexpected achievement of that individual effect u_n. To illustrate,

where a function depends on the effect $a \wedge b$, there will be unexpected consequences u_a and u_b.

A.2 States of device functions

As a device function f depends on two Boolean expressions, it can be in one of four states. These are termed inactive, In(f); failed, Fa(f); unexpected, Un(f) and achieved, Ac(f). These are defined in terms of the trigger and effect expressions as follows.

$$\mathrm{In}(f) \Leftrightarrow \neg t \wedge \neg e \tag{A.1}$$

$$\mathrm{Fa}(f) \Leftrightarrow t \wedge \neg e \tag{A.2}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg t \wedge e \tag{A.3}$$

$$\mathrm{Ac}(f) \Leftrightarrow t \wedge e \tag{A.4}$$

These are numbered 5.1 to 5.4 in Section 5.2. Additionally, the states triggered, Tr(f) and effective, (f) are defined in terms of the device function's trigger and effect respectively.

$$\mathrm{Tr}(f) \Leftrightarrow t \Leftrightarrow t \wedge (e \vee \neg e) \tag{A.5}$$

$$\mathrm{Ef}(f) \Leftrightarrow e \Leftrightarrow e \wedge (t \vee \neg t) \tag{A.6}$$

These are numbered 5.5 and 5.6 in Section 5.2. Notice that the state of a function can be defined in terms of these states, as follows.

$$\mathrm{In}(f) \Leftrightarrow \neg \mathrm{Tr}(f) \wedge \neg \mathrm{Ef}(f) \tag{A.7}$$

$$\mathrm{Fa}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \neg \mathrm{Ef}(f) \tag{A.8}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg \mathrm{Tr}(f) \wedge \mathrm{Ef}(f) \tag{A.9}$$

$$\mathrm{Ac}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \mathrm{Ef}(f) \tag{A.10}$$

These are equivalent to the definitions above and are numbered 5.7 to 5.10 in Section 5.2.

The design analysis report will include a reference to the state of any device function f that is failed or unexpected but not to a function that is inoperative or achieved.

$$\mathrm{R}(f) \text{ if } \mathrm{Fa}(f) \vee \mathrm{Un}(f) \tag{A.11}$$

It will also include a reference to the state of any individual effect e that is not

consistent with the expected state of the associated device function.

$$\mathrm{R}(e) \text{ if } (\mathrm{Tr}(f) \wedge \neg e) \vee (\neg \mathrm{Tr}(f) \wedge e) \tag{A.12}$$

These are numbered 5.11 and 5.12. in Section 5.7. Here an effect is taken to be any element of a Boolean expression that describes a function's required effects, so a design analysis report will include a reference to some absent or unexpected effect even where this does not amount to loss of a device function. In addition, the design analysis report will include consequences of failure of the function f and of unexpected achievement of the function or of an effect e as follows.

$$\mathrm{R}(c_f) \text{ if } \mathrm{Fa}(f) \text{ and if } (f \text{ includes } c_f) \tag{A.13}$$

This is numbered 7.5 in Section 7.2.

$$\mathrm{R}(u_f) \text{ if } \mathrm{Un}(f) \text{ and if } (f \text{ includes } u_f) \tag{A.14}$$

This is numbered 6.2 in Section 6.2.3. In rule A.13, the includes condition is required because the failed function might be an Operational Incomplete Function without its own consequences of failure, as described in Section 7.2. Where an effect e is achieved unexpectedly without unexpectedly achieving a function, or where a function has no unexpected consequences of its own, we can report u_e the consequences of the unexpected achievement of e.

$$\mathrm{R}(u_e) \text{ if } (\neg \mathrm{Tr}(f) \wedge e) \wedge (\neg \mathrm{Un}(f) \vee \neg(f \text{ includes } u_f)) \tag{A.15}$$

A.3 Subsidiary functions

Let f be a function composed of subsidiary functions a and b. These subsidiary functions are related using any Boolean operator $\otimes$, so f is composed of $a \otimes b$. It will also be associated with a description of purpose. The rule used for resolving the state of f is

$$\mathrm{Tr}(f) \text{ if } \mathrm{Tr}(a) \otimes \mathrm{Tr}(b) \tag{A.16}$$

$$\mathrm{Ef}(f) \text{ if } \mathrm{Ef}(a) \otimes \mathrm{Ef}(b) \tag{A.17}$$

These are 6.3 and 6.4 in Section 6.3. This allows the state of f to be defined in terms of its being triggered and effective, following rules A.7 to A.10. In other words, the trigger of f can be regarded as the combination of the triggers of a and b and the effect of f as the combination of the effects of a and b and those triggers

and effects are combined using the operator in the functional decomposition. This does restrict the use of subsidiary functions to cases where their trigger and effect expressions share an operator The use of incomplete functions, described below, circumvents this restriction.

The reporting of failure or unexpected achievement of functions follows rules A.11 and A.12 above. There are different rules for reporting the consequences of failure of functions. Let c_f be the consequences of failure of f and c_a and c_b be those of a and b respectively. The first rule is used if the subsidiary functions are combined using `AND` or `OR`. In these cases, if one but not both of the subsidiary function fails, the consequences of the failure of that subsidiary function are included, regardless of the state of the top function.

$$\mathrm{R}(c_f) \text{ if and only if } \mathrm{Fa}(a) \wedge \mathrm{Fa}(b) \tag{A.18}$$

$$\mathrm{R}(c_a) \text{ if } \mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b) \tag{A.19}$$

These rules are inappropriate where the subsidiary device functions are combined using `XOR` and the opposite approach has to be taken.

$$\mathrm{R}(c_f) \text{ if } (\mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b)) \vee (\neg\mathrm{Fa}(a) \wedge \mathrm{Fa}(b)) \tag{A.20}$$

These are numbered 6.5, 6.6 and 6.7 in Section 6.3. The reporting of consequences of unexpected achievement of a function (or effect) follow rules A.14 and A.15 above.

Where Purposive Incomplete Functions (PIFs) are used, so a device function f is composed of a trigger t and subsidiary PIFs combined using some Boolean operator $\otimes$, $p \otimes q$, then all these device functions will share the same value for the trigger and the state of f will depend on the truth of the effects of p and q.

$$\mathrm{In}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \neg(\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{A.21}$$

$$\mathrm{Fa}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \neg(\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{A.22}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge (\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{A.23}$$

$$\mathrm{Ac}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge (\mathrm{Ef}(p) \otimes \mathrm{Ef}(q)) \tag{A.24}$$

These are numbered 7.1 to 7.4 in Section 7.1. Note that these rules do not duplicate simply mapping a given trigger to two subsidiary functions a and b as if device function f is composed of $a \oplus b$ the trigger of both a and b will be true whenever either is triggered, so from rule A.16, f will never be triggered as it will never be

the case that only one of a and b will be triggered..

The rules for decomposition of a device function composed of Operational Incomplete Functions (OIFs) follows those for decomposition of a function composed of (complete) subsidiary functions. As an OIF has no description of purpose, and so no consequences of failure, the consequences of any OIF being in the failed state must be described in terms of the state of the function of which it is a component. This accounts for the "includes" condition in rule A.13 above.

The rules for Triggered Incomplete Functions (TIFs) are similar but the TIFs take the place of elements of the trigger expression of device function f. Let device function f be composed of TIFs $t \otimes u$ and an effect e.

$$\mathrm{In}(f) \Leftrightarrow \neg(\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \neg\mathrm{Ef}(f) \tag{A.25}$$

$$\mathrm{Fa}(f) \Leftrightarrow (\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \neg\mathrm{Ef}(f) \tag{A.26}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg(\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \mathrm{Ef}(f) \tag{A.27}$$

$$\mathrm{Ac}(f) \Leftrightarrow (\mathrm{Tr}(s) \otimes \mathrm{Tr}(u)) \wedge \mathrm{Ef}(f) \tag{A.28}$$

These rules are numbered 7.6 to 7.9 in Section 7.3. Note that these rules cannot be matched by sharing an effect between complete subsidiary functions as if one subsidiary function is effective, so will the other be so if the device function f is composed of $a \oplus b$, it will never be effective as it will never be the case that only one of a and b will be effective.

The rules for reporting of PIFs and TIFs follow those for reporting subsidiary functions, rules A.11 and A.12, and the rules for reporting of consequences of failure follow A.18, A.19 and A.20.

A.4 Sequential operators

To allow the description of functions that depend upon a sequence of effects the Functional Interpretation Language includes two sequential operators, SEQ and L-SEQ. These depend on a model of time that consists of intervals bounded by instants, as proposed in (Galton, 1990). Parts of the notation from that paper are used in this description. This model of time is consistent with the use of state charts to describe behaviour of a system or its components.

The sequence operator SEQ is true if the state described after the operator holds immediately after the present state of the system (defined in terms of the properties described in the functional description) ceases to hold. This can be

described using notation from (Galton, 1990). Let I_1 be an interval such that condition a holds throughout the interval, $\text{HOLDS_AT}(a, I_1)$, as Galton notates it, and I_2 be an interval such that $\text{HOLDS_AT}(b, I_2)$. Then,

$$a \text{ SEQ } b \text{ if } \text{HOLDS_AT}(a, I_1) \wedge \text{HOLDS_AT}(b, I_2) \wedge \text{MEETS}(I_1, I_2) \qquad \text{(A.29)}$$

Where MEETS follows the definition in (Allen, 1984),that the interval I_2 follows the interval I_1 with no interval separating them. The state preceding the first use of SEQ (or L-SEQ) in a functional description will not be defined as this will be the system state that was in effect before the triggering of that function. This state will not be known at model building time as a function might have alternative triggers, depending on the preceding system state, or the initial state might be affected by failure of some other function and so specification of the initial condition for some function would mean that failure of some preceding function would inevitably result in failure of the present function. SEQ is therefore treated as a unary operator, similar to the N (next time step) operator in Computational Tree Logic (CTL) (Emerson & Halpern, 1985).

The loose sequence operator L-SEQ is true if the state described after the operator holds some time after the present state of the system ceases to hold. That is to say that there can be an interval separating the described states. During this intervening interval the state of the described elements of the functional model is not defined. L-SEQ is similar to the F (some time in the future) operator in CTL and tense logic (Prior, 1957). It differs from SEQ in that the intervals when the states hold need not meet, as long as the second one follows the first eventually. It can be defined using Galton's HOLDS_IN operator, which is true if the state is present some of the time during some interval. So if I_1 is an interval such that $\text{HOLDS_IN}(a, I_1)$ and I_2 an interval such that $\text{HOLDS_IN}(b, I_2)$, then

$$a \text{ L-SEQ } b \text{ if } \text{HOLDS_IN}(a, I_1) \wedge \text{HOLDS_IN}(b, I_2) \wedge \text{MEETS}(I_1, I_2) \qquad \text{(A.30)}$$

As discussed in Chapter 8, this means the state of the system is undefined during some intermediate interval. L-SEQ has therefore to be used with some caution and can be replaced by a sequence of SEQ states so the intermediate states are defined.

Where SEQ (or L-SEQ) is used to combine subsidiary functions, rules A.16 and A.17 still hold.

The CYCLE and NEW-CYCLE operators and temporal constraints (BEFORE and AFTER) are not really susceptible to formal description. Informal definitions of these appear in Appendix B.

Appendix B

Notation used for functional description

This appendix contains a brief summary of the textual notation used for the Functional Interpretation Language described in this work, and as used throughout the thesis. The conventions used herein are also described.

B.1 The language

Here, the keywords used in the Functional Interpretation Language, together with those of the associated external descriptions, are listed, grouped according to their use.

B.1.1 Labels

This lists the keywords used to label different models and the components of these models.

`ACHIEVES` Used in a functional description to label the purpose that function fulfils, indicates the description of purpose.

`BY` Used to label the recogniser for a function, separating the recogniser from the ACHIEVES clause.

`FUNCTION` Used to label a functional description, as opposed to either an incomplete function or a description of purpose.

`OIF` Labels an "operational incomplete function", one that is a mapping between trigger and effect, with no distinct purpose.

`PIF` Labels a "purposive incomplete function", one that maps between an effect and a purpose, but shares a trigger with other elements of a top level function.

`PURPOSE` Labels a description of purpose (teleological model).

`TIF` Labels a "triggered incomplete function", that is one that maps a trigger to a purpose, allowing representations of functions that share an effect, whose purpose is defined by the trigger.

B.1.2 Operators

The conventional logical operators are used, with their usual meanings, so are not listed here.

`CYCLE` Indicates the start of a cyclical sequence of states that will continue until some event causes the cycle to be broken.

`L-SEQ` The system should enter the state following the operator at some time after the preceding state, as opposed to immediately after.

`NEW-CYCLE` Marks the end of a cycle. On reaching this, the function is expected to return to the state described after the corresponding `CYCLE`.

`SEQ` The system should enter for following the state immediately after the preceding state, so all expected effects should start simultaneously.

`TRIGGERS` Relates the (preceding) trigger expression to the effect expression of a function. The function is said to be achieved if both are true.

B.1.3 Functional states and relations

The four possible functional states are defined in terms of the truth of a function's trigger and effect, as discussed in Section 5.2. They are:-

`INOPERATIVE` A function's trigger and effect are both false, so the function is uncalled for.

`ACHIEVED` A function's trigger and effect are both true, so the function is achieved correctly and its purpose is being fulfilled.

`FAILED` A function's trigger is true but its effect is false, so the function is failing to fulfil its purpose even though it should be.

`UNEXPECTED` A function's trigger is false but its effect is true, so the effect is unnecessary.

It will be seen that the first two of these states are consistent with correct behaviour of the system, the other two are not. In addition to these states, two 'partial states' are defined, useful in specifying dependencies between functions. These are:-

`TRIGGERED` Simply means a function's trigger is true, so is equivalent to achieved OR failed.

`EFFECTIVE` A function's effect is true, so this is equivalent to achieved OR unexpected.

As all these states trace are defined in terms of truth of trigger and / or effect, there is no ambiguity.

B.1.4 Temporal constraints

These are used to specify that the expression to which the temporal constraint applies must be achieved within the specified time, measured from the preceding effective state (starting, typically) with the trigger becoming true).

`AFTER` Used to label the time before which the associated state should not become true, so specifies a minimum delay.

`BEFORE` Used to specify a time before which the associated state should become true, so specifies a deadline.

B.1.5 Other keywords

These include the items included in a description of purpose and / or the mapping between a functional description and the system model.

`DESCRIPTION` Labels the description of some purpose with which a function is associated.

`DETECTION` The value from 1 to 10 given for the likelihood that a function's failure or an unexpected effect will be noticed.

`FAILURE_CONSEQUENCE` Labels the description of the consequences of failure to fulfil the purpose with which the label is associated.

`IMPLEMENTS` Shows the mapping between a system property and the functional element (trigger or effect) with which it is associated.

`SEVERITY` The value from 1 to 10 given to show the seriousness of the consequences of a failure to achieve some purpose or of an unexpected effect.

`UNEXPECTED_CONSEQUENCE` Labels the textual description of the consequence of some effect (or function) being achieved unexpectedly.

B.2 Conventions used in this thesis

In the thesis, all quotations of or from a functional description, or other aspect of the Functional Interpretation Language, have been indicated by the use of a `fixed width font`. In addition, key words (including the conventional logic operators) have been placed in block capitals.

Text for quoting in a design analysis report, typically the consequences part of a description of purpose, are in quotation marks, in the same way as a string literal in a programming language. It is assumed that these quotes would not be part of the text, and will not appear in the report. This is also similar to their use in programming languages.

Appendix C

Common elements of diagrams

As there are several similar diagrams, it was felt worth including a key to the symbols and elements common to these diagrams.

The symbols used in the model relationship diagrams used in comparing approaches to and uses of functional modelling in Chapter 4 are shown in Figure C.1.

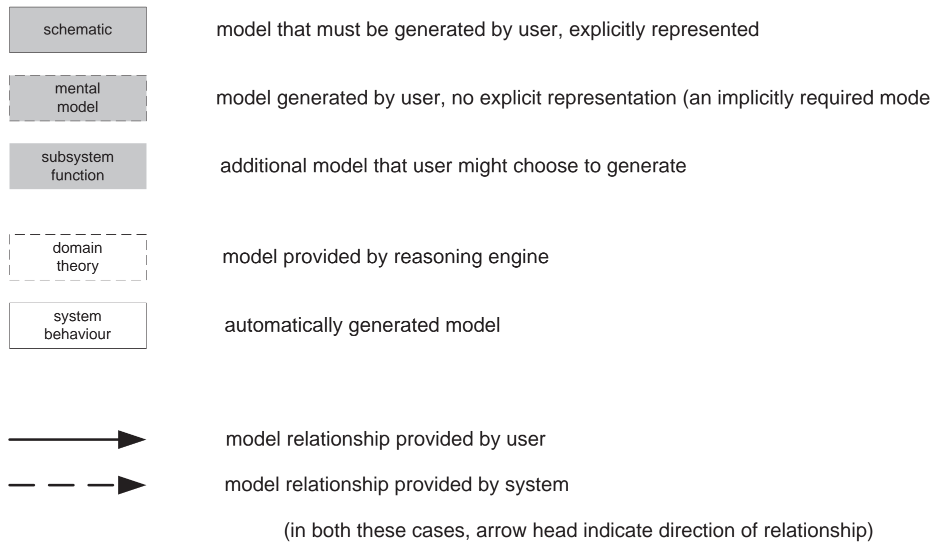


Figure C.1: Common symbols in the model relationship diagrams

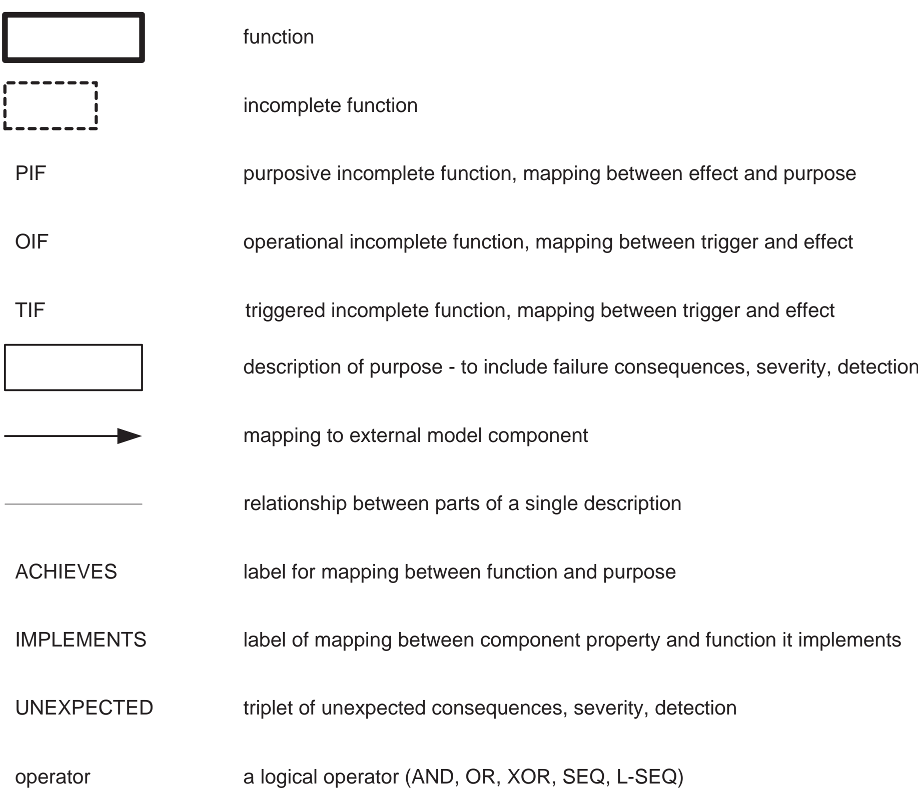


Figure C.2: Common symbols in the function composition diagrams

The symbols used in the diagrams used in Chapter 6, to illustrate functional decomposition are shown in Figure C.2. It should be noted that these diagrams are not intended to constitute a formal visual notation, they are simply intended to illustrate features of the language.

In these diagrams, an attempt has been made to arrange components as consistently as possible with the layout of the model relationship diagrams, so the description of purpose is shown to the right and mappings between the system model and the functional model to the left. While a simple functional decomposition (in terms of triggers and effects) is shown using plain lines, arrows are used to indicate that subsidiary functions (whether complete of incomplete) will be separate models, available for reuse independently of the present top level function. Any items at either end of an arrow are (or can be considered to be) separate parts of the model, stored independently. Naturally, whether or not this is actually the case might depend on the user's choice between reusability of models against the resulting complication of the model library. How this storage is managed is also dependent on the implementation. It might be done with a file hierarchy as suggested at the end of Chapter 11 or a relational database might possibly be used.

Appendix D

Description of functional decomposition tables

This appendix describes the notation used in the tables that illustrate the relationship between the logical basis of the functional language and the generated text in a resulting design analysis report, and especially to illustrate the functional decompositions. Tables using this notation appear in Chapters 5, 6 and 7. Because of their appearance in different places of the thesis and to avoid breaking up the text around the earlier appearances of such tables, it was decided to place a detailed description of them here, for reference.

Table D.1 is an example of such a table. This example table contains selected

function *C1*			function *C2*			*F* (*C1* AND *C2*)			generated text	consequences
t1	*e1*	state	*t2*	*e2*	state	t	e	state		
F	F	In	F	F	In	**F**	**F**	**In**	(no entry)	
F	F	In	T	T	Ac	**F**	**F**	**In**	Function *C2* achieved	
F	F	In	T	F	Fa	**F**	**F**	**In**	Function *C2* failed because expected effect *e2* absent	Fa(*C2*)
F	F	In	F	T	Un	**F**	**F**	**In**	Function *C2* achieved unexpectedly because unexpected effect *e2* present	Un(*C2*) or Un(*e2*)
F	T	Un	F	T	Un	**F**	**T**	**Un**	Function *F* achieved unexpectedly because functions *C1* and *C2* achieved unexpectedly	Un(*F*) or max (Un(*C1*), Un(*C2*))

Table D.1: Example functional decomposition table

rows from Table 6.2 in Chapter 6. In the headings, “state” has been used to label

the functional state of a function as shown in Table 5.1 on page 92. those columns can therefore have any of the four values:-

Inoperative (In) Both trigger and effect are false, so function is uncalled for and (correctly) not achieved.

Achieved (Ac) Both trigger and effect are true and function is achieved as expected (though possibly as a result of unexpected internal behaviour).

Failed (Fa) Trigger is true but effect false, so triggering the function has not resulted in the expected effect.

Unexpected (Un) Trigger is false but effect true, so the effect is present despite being uncalled for.

As these values are defined in terms of the truth of trigger and effect, including the values of trigger, effect and state for the subsidiary functions (*C1* and *C2*) is tautologous. It was felt that the full description would clarify the relationships between triggers, effects and states in the decompositions, however.

The heading for trigger is abbreviated to 't' and for effect to 'e'. Where a specific trigger or effect might appear in the resulting design analysis report (shown in the generated text column), it has been italicised and where necessary identified with a suffix, so *e1* is the effect associated with the first child function. Function names (that might also appear in a resulting report) are similarly italicised, but are in capitals.

As triggers and effects are Boolean expressions, they have the values true (T) or false (F). It is to reduce the likelihood of confusion with the value 'false' that 'failed' is abbreviated to Fa and this has led to the use of two letter abbreviations for the other functional states throughout the thesis.

The layout of this side of the tables showing purposive incomplete functions (PIFs) is slightly different, as shown in Table D.2. Here the trigger is common to the top level function (called *F*) and both of the subsidiary PIFs. Each PIF has columns for its effect and state. There is a column for the effect expression of the top level function, derived from the effects of the PIFs, labelled e*F*, and one for its state. The top level function is labelled with the relation between its subsidiary PIFs in the header.

The entry in the "generated text" column is a brief example of what might appear in the resulting report, using the names identified in the table headings. It might, of course, be the case that the actual text is more detailed than appears here, so that instead of the entry in the last line of the table above, a report might include

function F ($P1$ AND $P2$)							generated text	consequences
t	$P1$		$P2$					
	$e1$	state	$e2$	state	eF	state		
F	F	In	F	In	**F**	**In**	(no entry)	
F	T	Un	F	In	**F**	**In**	Function $P1$ achieved unexpectedly because unexpected effect $e1$ present	Un($P1$) or Un($e1$)
F	T	Un	T	Un	**T**	**Un**	Function F achieved unexpectedly because functions $P1$ and $P2$ achieved unexpectedly	Un(F) or max (Un($P1$), Un($P2$))

Table D.2: Example of decomposition using PIFs

> Function F achieved unexpectedly because function $C1$ achieved unexpectedly because unexpected effect $e1$ was present and function $C2$ achieved unexpectedly because unexpected effect $e2$ was present.

In these tables, the functional labels have been used, but it would be feasible to use the linked system properties, traced from the mappings between functional and system models, in a design analysis report.

The consequences are assumed to be a set of textual description, value for severity and value for detection, associated with a function's description of purpose (though here simply associated with the function itself) or with an effect. This should be associated with the mapping between the effect and the system model, of course, but the functional labels themselves are used in the tables, for simplicity. The entry Fa($C2$) means the consequences associated with the failure of (the description of purpose associated with) function $C2$. As a description of purpose might or might not have unexpected consequences, this raises the possibility of alternative consequences associated with unexpected achievement of a function's effect(s). For example, in the fourth line of Table D.1, "U($C2$) or U($e2$)" means that if there are unexpected consequences associated with $C2$ include them, if not use the (required) ones for the effect $e2$. These might differ, of course, as $C2$ might have several effects combined using a Boolean expression.

Where there are several possible failures whose consequences might be listed, different approaches are possible, as discussed in the text. In the tables it is assumed that the more severe of the consequences are listed, indicated by 'max'. In practice, this is more complex than the tables suggest, because the consequences with the higher value for severity will be used (unless both are listed in the report, of course) but it is arguably more appropriate to use the lower value for detection, as this shows how likely the fault will be noticed by the operator, and which might not belong to the same set of consequences. In the tables the columns for

consequences, severity and detection, as illustrated in Table 5.2 on page 110, are treated as one, to save space and simplify the relationships.

Glossary

Behavioural knowledge Knowledge of what happens within a device.

Dependent function Any function that depends on the state of some other system function for its trigger.

Detection A value between 1 and 10 indicating the likelihood that some failure will be detected before any consequences arise.

FMEA Failure Mode Effects Analysis, a design analysis that establishes the effects on a system of failures to components in that system.

FTA Fault Tree Analysis, a design analysis in which the result is a report that relates effects (symptoms) back to candidate causes.

Functional (design) analysis A term used to group those design analysis tasks that are concerned with whether a system fulfils its intended function, either when working correctly or under a component failure.

Functional description A representation of an individual function of a system that achieves one of the system's intended purposes.

Functional knowledge Knowledge of how a device achieves its purpose. More formally defined as

> An object O has a function F if it achieves an intended goal by virtue of some external trigger T resulting in the achievement of an external effect E.

Functional model The collection of one or more functional descriptions associated with some system.

Occurrence A value between 1 and 10 indicating the likelihood of the cause of some failure.

Ontology The nature of the model used as the basis for model based reasoning. Typically concerned with whether the model is built using knowledge of components, processes or constraints.

Operational incomplete function A representation of a function that only represents the relation between trigger and effect. It should only be used as part of a functional decomposition to ensure triggers are associated with the appropriate effects.

Prime function Some function that fulfils a purpose of the system. Used specifically to refer to a function upon which some other function depends.

Purposive incomplete function A representation of function that only represents the mapping between effect and purpose. It should only be used as part of a functional decomposition.

Risk Priority Number (RPN) A value indicating the seriousness of a system failure and therefore how much priority should be given to eliminating the failure. It is the product of values for severity, detection and occurrence.

SCA Sneak Circuit Analysis, an electrical design analysis that ensures that no unexpected effects occur as a result of combinations of switch settings.

Severity A value between 1 and 10 indicating the seriousness of consequence of some failure.

Structural knowledge Knowledge of the components (or processes) that make up a device and the connections between them.

Teleological knowledge Knowledge of the purpose of a device. In the representation of function proposed herein, it includes a description of the consequences of a function's failure to achieve the purpose.

Triggered incomplete function A representation of function that relates a trigger to its intended purpose. It is used where an effect is common to several triggers, which determine the purpose of achieving the effect.

References

Allen, James F. 1984. Towards a general theory of action and time. *Artifial intelligence*, **23**(2), 123–154.

Baher, Julie. 1999. Articulate virtual labs in thermodynamics education: A multiple case study. *Hournal of engineering education*, October, 429–434.

Bell, Jonathan, & Snooke, Neal A. 2004. Describing system functions that depend on intermittent and sequential behavior. *Pages 51–57 of: Proceedings 18th international workshop on qualitative reasoning, QR2004*.

Bell, Jonathan, Snooke, Neal A., & Price, Christopher J. 2005a. Functional decomposition for interpretation of model based simulation. *Pages 192–198 of: Proceedings of 19th international workshop on qualitative reasoning (QR-05)*.

Bell, Jonathan, Snooke, Neal A., & Price, Christopher J. 2005b. A language for functional interpretation of model based simulation. *Pages 1547–1548 of: Proceedings of international joint conference on artificial intelligence (IJCAI 2005)*.

Bosch. 1991. *CAN specification version 2.0*. Robert Bosch GmbH. Bosch CAN website is at http://www.can.bosch.com/.

Bouwer, Anders, & Bredeweg, Bert. 2001. VisiGarp: Graphical representation of qualitative simulation models. *Pages 142–149 of: Proceedings 15th international workshop on qualitative reasoning, QR '01*.

Bowles, John B., & Wan, C. 2001. Software failure modes and analysis for a small embedded control system. *Pages 1–6 of: Proceedings annual reliability and maintainability symposium*.

Bredeweg, Bert. 1992. *Expertise in qualitative prediction of behaviour*. Ph.D. thesis, University of Amsterdam.

Chandrasekaran, B. 2005. Repersenting function: Relating functional rpepresentation and functional modeling research streams. *Artificial intelligence for engineering design, analysis and manufacturing*, **19**, 65–74.

Chandrasekaran, B., & Josephson, John R. 1996. Representing function as effect: Assigning functions to objects in context and out. *In: Proceedings of american association for artificial intelligence.*

Chandrasekaran, B., & Josephson, John R. 2000. Function in device representation. *Engineering with computers*, **16**(3–4), 162–177.

Chandrasekaran, B., Goel, A. K., & Iwasaki, Y. 1993. Functional representation as design rationale. *IEEE computer*, **26**(1), 48–56.

Chittaro, Luca, & Kumar, Amruth N. 1998. Reasoning about function and its applications to engineering. *Artificial intelligence in engineering*, **12**(4), 331.

Chittaro, Luca, Guida, Giovani, Tasso, Carlo, & Toppano, Elio. 1993. Functional and teleological knowledge in the multimodeling approach to reasoning about physical systems. *IEEE transactions on systems, man and cybernetics*, **23**(6), 1718–1751.

Chittaro, Luca, Tasso, Carlo, & Toppano, Elio. 1994. Putting functional knowledge on firmer ground. *Applied artificial intelligence*, **8**, 239–258.

David, Jean-Marc, & Krivine, Jean-Paul. 1986. Reasoning from structure and behavior: Four relevance criteria. *Pages 120–127 of: European conference on artificial intelligence*, vol. 2.

Davis, Randall, & Hamscher, Walter. 1988. Model-based reasoning: Troubleshooting. *Pages 297–346 of:* Shrobe, H. (ed), *Exploring artificial intelligence.* Morgan Kaufmann.

de Kleer, Johan. 1984. How circuits work. *Artificial intelligence*, **24**, 205–280.

Emerson, E. A., & Halpern, J. Y. 1985. Design procedures and expressiveness in the temporal logic of branching time. *Journal of computer and system sciences*, **30**(1), 1–24.

Falkenhainer, Brian, & Forbus, Kenneth D. 1991. Compositional modelling: Finding the right model for the job. *Artificial intelligence*, **51**, 95–143.

Forbus, Kenneth D. 1984. Qualitative process theory. *Artificial intelligence*, **24**, 85–168.

Forbus, Kenneth D. 1988. Qualitative physics: Past, present, and future. *Pages 239–296 of:* Shrobe, H. E., & for Artificial Intelligence, American Association (eds), *Exploring artificial intelligence: Survey talks from the national conferences on artificial intelligence*. San Mateo, CA: Kaufmann.

Forbus, Kenneth D. 1997. Using qualitstive physics to create articulate educational software. *IEEE expert*, 32–41.

Franke, David W. 1991. Deriving and using descriptions of purpose. *IEEE expert: Intelligent systems and their application*, **6**(2), 41–47.

Franke, David W. 1993. Clarifying terminology in functional reasoning. *Page iv of: Reasoning about function: Workshop notes of AAAI-93*.

Galton, Antony. 1990. A critical examination of Allen's theory of action and time. *Artificial intelligence*, **42**(2–3), 159–188.

Gerevini, Alfonso, & Schubert, Lenhart. 1995. Efficient algorithms for qualitative reasoning about time. *Artificial intelligence*, **74**(2), 207–248.

Gero, John. 1990. Design prototypes: A knowledge representation schema for design. *AI magazine*, **11**(4), 26–36.

Goddard, Peter. 2000. Software FMEA techniques. *Pages 118–123 of: Procceedings of the annual reliability and maintainability symposium*. IEEE.

Gunzert, Michael, & Nägele, Andreas. 1999. Component-based development and verification of safety critical software for a brake-by-wire system with synchronous software components. *Pages 134– of: Pdse*.

Harel, David. 1987. Statecharts: A visual formalism for complex systems. *Science of computer programming*, **8**(June), 231–274.

Hawkins, P. G., & Woollons, D. J. 1998. Failure modes and effects analysis of complex engineering systems using functional models. *Artificial intelligence in engineering*, **12**(4), 375–397.

Hayes, P. 1985. The second naive physics manifesto. *In:* Hobbs, & Moore (eds), *Formal theories of the commonsense world*. Ablex Publishers.

Iwasaki, Yumi, Fikes, R., Vescovi, M., & Chandrasekaran, B. 1993. How things are intended to work: Capturing functional knowledge in device design. *Pages 1516–1522 of: Proceedings of 13th international joint conference on artificial intelligence*.

Iwasaki, Yumi, Vescovi, M., Fikes, R., & Chandrasekaran, B. 1995. Causal functional representation language with behavior-based semantics. *Applied artificial intelligence*, **9**(1), 5–31.

Kaindl, Hermann. 1993. Distinguishing between functional and behavioral models. *Pages 50–52 of: Reasoning about function: Workshop notes of AAAI-93.* AAAI.

Kampis, G. 1987. Some problems of system descriptions I: Function. *International journal of general systems*, **13**, 143–156.

Keuneke, Anne M. 1991. Device representation: The significance of functional knowledge. *IEEE expert*, **6**, 22–25.

Keuneke, Anne M., & Allemang, Dean. 1988. *Understanding devices: Representing dynamic states*. Tech. rept. 88-AK-DYNASTATES. Ohio State University. From the Laboratory for Artificial Intelligence Research, Department of Computer Science.

Kitamura, Yoshinobu, Sano, Toshinobu, Namba, Kouji, & Mizoguchi, Riichiro. 2002. A functional concept ontology and its application to automatic identification of functional structures. *Advanced engineering informatics*, **16**, 145–163.

Kuipers, Benjamin J. 1986. Qualitative simulation. *Artificial intelligence*, **29**, 289–338.

Larsson, J. E. 1996. Diagnosis based on explicit means-ends models. *Artificial intelligence*, **80**(1), 29–93.

Lee, Mark H. 1999a. Qualitative circuit models in failure analysis reasoning. *Artificial intellligence*, **111**, 239–276.

Lee, Mark H. 1999b. Qualitative modelling of linear networks in ECAD applications. *Pages 146–152 of: Proceedings 13th international workshop on qualitative reasoning, QR '99.*

Lee, Mark H., & Ormsby, Andrew R. T. 1991. A qualitative circuit simulator. *In: Second annual conference on AI simulation and planning in high autonomy systems.* IEEE.

Lee, Mark H., Bell, Jonathan, & Coghill, George Macleod. 2001. Ambiguities and deviations in qualitative circuit analysis. *Pages 51–58 of: Proceedings 15th international workshop on qualitative reasoning, QR '01.*

Leitch, R., Shen, Q., Coghill, G. M., & Chantler, M. 1999. Choosing the right model. *Pages 435–449 of: IEE proceedings d, control theory and applications*, vol. 146.

Lind, M. 1994. Modelling goals and functions of complex industrial plants. *Applied artificial intelligence*, **8**, 259–283.

Loganantharaja, Rasia. 1993. Representation of functional knowledge. *Pages 102–107 of: Reasoning about function: Workshop notes of AAAI-93*. AAAI.

Machado, Vânia Bessa, & Bredeweg, Bert. 2001. Towards interactive tools for constructing articulate simulations. *Pages 98–104 of: Proceedings 15th international workshop on qualitative reasoning, QR '01*.

Maier, T. 1995. FMEA and FTA to support safe design of embedded software in safety-critical systems. *In: CSR 12th annual workshop on safety and reliability of software based systems*.

Manzone, Alberto, Pincetti, Alessandro, & de Costantini, Diego. 2001. Fualt tolerant automotive systems: An overview. *In: Proceedings of the seventh IEEE international on-line testing workshop*.

Mauss, Jakob, & Neumann, Bernd. 1996. Qualitative reasoning about electrical circuits using series-parallel-star trees. *Pages 147–153 of: Proceedings 10th international workshop on qualitative reasoning, QR-96*.

McManus, Alex, Price, Christopher J., Snooke, Neal A., & Joseph, Richard. 1999. Design verification of automotive electrical circuits. *In: Proceedings 13th international workshop on qualitative reasoning*.

Milde, Heiko, Hotz, Lothar, Kahl, Jörg, & Wessel, Stephanie. 1999. Qualitative analysis of electrical circuits for computer-based diagnostic decision tree generation. *Pages 204–210 of: Proceedings conference on diagnosis, dx99*.

Navinchandra, Dundee, & Sycara, Katia P. 1989. Integrating case-based reasoning and qualitative reasoning in design. *In:* Gero, J. (ed), *AI in design*. Computational Mechanics.

Papadopoulos, Yiannis, McDermid, John, Mavrides, Androcles, Scheidler, Christian, & Maruhn, Matthias. 2001. Model-based semiautomatic safety analysis of programmable systems in automotive applications. *Pages 53–57 of: Proceedings of ADAS2001, the international conference on advanced driver assistance systems*. IEEE Publications.

Price, Christopher J. 1996. Effortless incremental design FMEA. *Pages 43–47 of: Proceedings of annual reliability and maintainability symposium.*

Price, Christopher J. 1998. Function-directed electrical design analysis. *Artificial intelligence in engineering*, **12**(4), 445–456.

Price, Christopher J. 2000. AutoSteve: automated electrical design analysis. *Pages 721–725 of: Proceedings ECAI-2000.*

Price, Christopher J. 2002 (March). Incremental automated diagnosis. *Pages 45–50 of: Proceedings of the AAAI spring symposium on information refinement and revision for decision making.*

Price, Christopher J., & Hunt, John E. 1989. Augmenting qualitative diagnosis. *In: Proceedings of the 6th alvey DKBS workshop.* IEE.

Price, Christopher J., & Pugh, David. 1996. Interpreting simulation with functional labels. *Pages 198–204 of: Proceedings 10th international workshop on qualitative reasoning.* AAAI Press.

Price, Christopher J., Snooke, Neal, & Landry, J. 1996a. Automated sneak identification. *Engineering applications of artificial intelligence*, **9**(4), 423–427.

Price, Christopher J., Wilson, Myra S., & Cain, C. 1996b. Automotive diagnosis using generated fault trees. *Pages 161–167 of: Applications and innovations in expert systems iv (proceedings expert systems 96).*

Price, Christopher J., Wilson, Myra S., Timmis, Jonathan, & Cain, C. 1996c (September). Generating fault trees from FMEA. *Pages 183–190 of: 7th international workshop on principles of diagnosis.*

Price, Christopher J., Snooke, Neal, Pugh, David, Hunt, John E., & Wilson, Myra S. 1997. Combining functional and structural reasoning for safety analysis of electrical designs. *The knowledge engineering review*, **12**(3), 271–287.

Price, Christopher J., Snooke, Neal A., & Lewis, Stuart D. 2003. Adaptable modeling of electrical systems. *Pages 147–153 of:* Salles, Paulo, & Bredeweg, Bert (eds), *Proceedings of 17th international workshop on qualitative reasoning (QR2003).*

Prior, Arthur N. 1957. *Time and modality.* Oxford: Clarendon Press.

Pugh, David, & Snooke, Neal A. 1996. Dynamic analysis of qualitative circuits. *Pages 37–42 of: Proceedings annual reliability and maintainability symposium.*

Pugh, David, Price, Christopher J., & Snooke, Neal A. 1995. Practical applications for multiple models - the need for simplicity and reusability. *Applications and innovations in expert systems iii*, 277–291.

Quarles, T., *et al.* 1980. *Spice version 3: Users' guide.* On line version available at http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE.

Raiman, Olivier. 1986. Order of magnitude reasoning. *In: AAAI-86 american association for artificial intelligence.*

Raiman, Olivier. 1991. Order of magnitude reasoning. *Artificial intelligence*, **51**, 11–38.

Saber. 1996. *Saber users' guide.* Analogy, inc.

Sasajima, M., Kitamura, Y., Ikeda, M., & Mizoguchi, R. 1995. FBRL: A functional and behavior representation language. *In: Proceedings international joint conference on artificial intelligence (IJCAI-95).*

Savakoor, D. S., Bowles, J. B., & Bonnell, R. D. 1993. Combining sneak circuit analysis and failure modes and effects analysis. *Pages 199–205 of: Proceedings annual reliability and maintainability symposium.*

Sembugamoorthy, V, & Chandrasekaran, B. 1986. Functional representation of devices and compilation of diagnostic problem-solving systems. *Pages 47–73 of:* Kolodner, Janet L., & Riesbeck, Christopher K. (eds), *Experience, memory and reasoning.* Erlbaum.

Snooke, Neal A. 1999. Simulating electrical devices with complex behaviour. *AI communications*, **12**(1,2), 45–58.

Snooke, Neal A. 2004. Model-based failure mode and effects analysis of software. *Pages 221–226 of: Proceedings of 15th international workshop on the principles of diagnosis (DX04).*

Snooke, Neal A., & Bell, Jonathan. 2002. Abstracting automotive system models from component-based simulation with multi level behaviour. *Pages 151–160 of: Sixteenth international workshop on qualitative reasoning (QR02).*

Snooke, Neal A., & Price, Christopher J. 1998. Hierarchical functional reasoning. *Knowledge-based systems*, **11**(5–6), 301–309.

Sticklen, Jon, Goel, A., Chandrasekaran, B., & Bond, W. E. 1989. Functional reasoning for design and diagnosis. *In: Proceedings model based diagnosis international workshop (DX-89).*

Sticklen, Jon, Kamel, Ahmed, & Bond, William E. 1991. Integrating quantitative and qualitative computations in a functional framework. *Engineering applications of artificial intelligence*, **4**(1), 1–10.

Tuttle, Kenneth L., & Wu, Chih. 2001. Intelligent computer assisted instruction in thermodynamics at the U. S. Naval Academy. *Pages 105–112 of: Proceedings 15th international workshop on qualitative reasoning (QR '01).*

Umeda, Yasushi, & Tomiyama, Tetsuo. 1993. A CAD for functional design. *Pages 172–179 of: Reasoning about function: workshop notes of AAAI-93.*

Umeda, Yasushi, Kondoh, Sinsuke, Shimomura, Yoshiki, & Tomiyama, Tetsuo. 2005. Development of design methodolgy for upgradable products based on function-behavior-state modeling. *Artificial intelligence for engineering design, analysis and manufacturing*, **19**, 161–182.

van Wie, Michael, Bryant, Cari R., Bohm, Matt R., McAdams, Daniel A., & Stone, Robert B. 2005. A model of function-based representations. *Artificial intelligence for engineering design, analysis and manufacturing*, **19**, 89–111.

Wirth, Rüdiger, & O'Rorke, Paul. 1993. Representing and reasoning about functions for failure modes and effects analysis. *Pages 188–193 of: Reasoning about function: Workshop notes of AAAI-93.* AAAI.

Wood, William H., Dong, Hui, & Dym, Clive L. 2005. Integrating functional synthesis. *Artificial intelligence for engineering design, analysis and manufacturing*, **19**, 183–200.