Acknowledgements

I wish to thank first of all my Director of Studies, Kate Norrie, for her expertise and continual support through the duration of my research. Thanks also to my other two supervisors: Phil Curran, for providing me with insights into the medical scene and a useful case study; and to Peter Forte, for keeping up my enthusiasm during the rather lengthy process of writing up. Additional thanks to the other researchers in the CSES Research Lab, the School secretaries and the technical support. I also acknowledge the financial support of the EPSRC who provided me with a research studentship.

I would like regarding the technical content to make special mention of Andrzej Wardziński and Glenn Bruns for encouraging me in the lines of research I most wanted to pursue. Also, I'm grateful to Alfred Crabtree, FIEE, for instilling in me a greater awareness of the wider setting.

Finally, the research would not have been completed without moral support, especially from my parents, and also Sandy Martin, Thaer Sabri plus anyone else who has helped me on the way.

Abstract

The Use of Formal Methods for Safety-Critical Systems

An investigation is presented into the use of formal methods for the production of safety-critical systems with embedded software. New theory and procedures are tested on an industrial case study, the formal specification and refinement of a communications protocol for medical devices (the Universal Flexport protocol ©).

On reviewing the current literature, a strong case emerges for grounding any work within an overall perspective that integrates the experience of safety engineering and the correctness of formal methods. Such a basis, it is argued, is necessary for an effective contribution to the delivery with assurance of life-critical software components.

Hence, a safety-oriented framework is proposed which facilitates a natural flow from safety analysis of the entire system through to formal requirements, design, verification and validation for a software model undergoing refinement towards implementation. This framework takes a standard safety lifecycle model and considers where and how formal methods can play a part, resulting in procedures which emphasise the activities most amenable to formal input.

Next, details of the framework are instantiated, based upon the provision of a common formal semantics to represent both the safety analysis and software models. A procedure, **FTBuild** , is provided for deriving formal requirements as part of the process of generating formalised fault trees. Work is then presented on establishing relations between formalised fault trees and models, extending results of other authors. Also given are some notions of (property) conformance with respect to the given requirements.

The formal approach itself is supported by the enhancement of the theory of conformance testing that has been developed for communication systems. The basis of this work is the detailed integration of already established theories: a testing system for process algebra (the Experimental System due to Hennessy and de Nicola) and a more general observation framework (developed by the LOTOSphere consortium). Notions of conformance and robustness are then examined in the context of refinement for the process algebra, (Basic) LOTOS, resulting in the adoption of the commonly accepted 'reduction' relation for which a proof is given that it is testable. Then a new algorithm is developed for a single (canonical) tester for reduction, which is unified in that it tests simultaneously for both conformance and robustness. It also allows, in certain cases, a straghtforward implementation as a Full LOTOS process with the ability to give some diagnostics in the case of failure. The text is supported by examples and some guidelines for use.

Finally, having established these foundations, the methodology is demonstrated on the Flexport protocol through two iterations of **FTBuild** which demonstrate how the activities of specification, safety analysis, validation and refinement are all brought together.

Contents

List of Figures

List of Tables

Chapter 1

Overview of Thesis

1.1 Aim and Objectives

The aim of this thesis is to investigate how the use of formal methods may be effectively realized for the production of safety-critical systems.

The objectives are:

1. To propose a framework for the overall system that enables the role of formal methods to become clear;
2. To establish closer links between safety analysis and formal representations;
3. To enrich formal theory itself in response to typical safety requirements;
4. To validate the theory and methods in an industrial case study

A particular feature of this thesis is that safety-related properties in the formal context are generated in a manner that is designed to enable justifiable assurance: they should be easily traceable back to the safety analysis for the system.

1.2 Structure of thesis

The objectives are tackled as follows.

The initial sections of Chapter 2 present an overview of the use of formal methods for safety critical systems. They start by describing how issues such as reliability and safety have been treated in engineering as a whole, methods which have gained a good deal of maturity. After this are discussed some of the current views on and approaches to the production of safety-related software, including the use of standard models. Using these

models, it is shown how safety concepts may be highlighted and where and how formal methods may play a key role. Some motivating examples are provided to make explicit some of the technical issues.

In chapter 3 a proposal is presented, setting out the motivation for and aims of a safety-oriented framework for the refinement of formal specifications. This is based on the consideration of a standard Safety Lifecycle Model and how it can support the formal activities. To facilitate more effective use of formal methods, some essential 'project hygiene' is incorporated within the lifecycle model, including aspects of Configuration Management plus the indication of how tools may be used. From the perspective of formal methods, we discuss how their use necessarily raises issues about the framework itself. Accordingly, to support the central construction role of formal methods, we discuss some formalisation of aspects of the design cycle with particular regard to their relationship to formal models.

We elaborate on this aspect of the framework in chapter 4, focusing on methods to integrate specific safety analysis techniques with the formal development, showing how formalising the analysis can aid in understanding the system. There are a range of techniques available, from which we choose to concentrate on one particular type, *fault tree analysis*, which has received relatively more attention that others, though this is still rather modest. Indeed, on reviewing the state of the art, it is evident that most of the work has tended to concentrate on providing semantics for the trees.

Here, a procedure **FTBuild** is provided which is centred around the formalisation of a fault tree. It develops the tree step by step and independently of the semantics chosen and shows how this can be tied into the development of models. In particular, the formalised trees are used as the basis for the generation of safety requirements for models, which we support by the listing of some issues and criteria. Having established the requirements, we then build on some work by Bruns and Anderson [Bru93], by introducing some new general relations for *property conformance* between a set of safety requirements derived from formalised fault trees and a model. These relations allow flexibility in the choice of requirements, to take account of the potential disparity in stage of development between the requirements derivation and the model construction. The requirements generated are in general any kind of predicate – they can be logical formulae for a given model, or relations between models. All we assume is that they all are semantically based in labelled transition systems.

Regarding the relations between models, there are a number of notions available which define what constitutes a valid refinement from one model to another. One of these notions is conformance, which has particular relevance in the field of communication pro-

tocols. This leads us to the consideration of valid refinements of safety-critical protocols, which require two properties to hold – conformance (implementing what has been prescribed) and robustness (not implementing what is disallowed). Further, since in practice an implementation cannot be examined internally, methods of testing have been developed to determine whether or not there is conformance on the basis of external observations – *conformance testing*.

There has been considerable work on conformance testing, but little on robustness, often in view of the state explosion problem. However, there are cases where there is not this problem, for instance when part of a protocol consists in effect of accepting only certain permutations on a finite alphabet – which is the case for the protocol we examine. As robustness is critical for safety, we investigate ways of testing for it, motivating a formal theory of robust conformance testing for process algebras, the subject of chapter 5.

The chapter starts with an introduction cum survey showing how notions of testing in engineering may be given meaning in a formal setting, specifically for the ISO specification language LOTOS [ISO89a]. This motivates the definition of an already existing observation framework which was developed as a task in the LOTOSphere project for conformance testing (Task 1.3, [ABe+90]). This framework underpins the subsequent work, which starts by showing in some detail how it may incorporate the Experimental System of Hennessy and De Nicola as detailed in [Hen88].

The main application of the theory is then presented, leading to an algorithm that is proposed for deriving a canonical tester for the commonly accepted notion of robust conformance called *reduction*. The tester is *unified* in that it tests simultaneously for robustness (trace inclusion) and conformance. Further, for a special case, a method is given for implementing the algorithm for its tester as a (Full) LOTOS specification which has the distinctiveness of providing special diagnostics in the case of failure. Finally, there is some discussion of the work in the light of practice, including the consideration of alternative formalisations of conformance, for which a new relation is suggested.

The work culminates in chapter 6 with an application of the methodology and theory to the modelling and analysis of the Universal Flexport protocol for medical devices. The protocol specifies how a third party device should be connected to a proprietary system, henceforth referred to simply as *Flexport*. Our project presents an instantiation of the general framework developed earlier, and operates according to a Configuration Management plan. The main task is to refine specifications in LOTOS of the link connection phase, all subject to version control. The process of refinement is centred around two iterations of **FTBuild** , so the methodology is requirements-driven, with detailed safety analysis

prompted by the fault trees.

Within this context are presented distinct approaches to verification and validation, which we contrast and compare. Among the verification tasks, we show the consistency of two distinct views (intensional and extensional) of the protocol specification and then the consistency of a further refinement which incorporates more detail. The validation requires proof of some safety related properties, particularly liveness. The methodologies we employ for the verification are based upon observation equivalence and the testing relations developed earlier; the validation is conducted through simulation, testing and model checking.

Finally, in chapter 7, we present a summary of our investigation and its contribution to the subject area, offering as the main outcome a refined perspective on how to produce systems in which we may be confident that they fulfil the safety requirements of users. This is concluded by some directions for future research – it is evident there is a great deal of scope.

Chapter 2

Introduction

2.1 Software in Safety-critical systems

2.1.1 The Role of Software in Safety-critical systems

Software is playing an increasingly important role in systems, most notably in embedded systems, where it is used to control machines. A growing number of these systems are safety-critical, where there is risk to life. When we make use of such systems, we trust that their risk has been minimized, so that the operation of the controlling software components are effectively safe. Not only that, we also expect them to be reliable, cost effective and possessing of many other attributes. Those responsible for designing and delivering safety-critical systems must ensure that they can satisfy these requirements and be able to demonstrate this, ideally in a manner accessible to all who use the system, often via public bodies such as standards authorities.

The urgency to address this area comes not least from fatalities that have been substantially due to software-related errors – for instance, overdoses from Therac-25, a linear accelerator for treating cancer through radiation [LT93] and the overshooting of the runway at Warsaw airport by an Airbus A320 [Mai94]. Other hazardous incidents such as the Ariane V rocket going off course may not have cost lives, but have certainly proved that design errors are expensive [Lio96].

This is a problem that is well known to other engineering disciplines and over the years experience has been accumulated to provide effective solutions: the technology has been available and it has been shown to work with a very high degree of confidence. Software, although a fairly recent phenomenon, is also classified as an engineering discipline, so it would seem natural that one can apply the insights and quickly surmount any hurdles. However, whilst it is true that engineering insights can and should be applied (as is the

contention of this thesis), it has become evident that software is in some ways another 'kettle of fish' altogether. In the next sections we discuss what the problems are, and the approaches to tackling them, leading onto the need for formal methods and how that may used to help the situation.

2.1.2 Terminology

We provide some definitions to establish clearly what is being discussed and analysed, starting with terminology covering any kind of system and then introducing some terms specific to software.

> *Safety* is a value judgement, perceived essentially as protection from loss (or injury), be it physical, social or environmental.
>
> A *hazard* is a set of conditions in which the protection is reduced, that is unsafe to some degree, and has an associated risk of loss.
>
> *Risk* is defined in terms of three factors: the likelihood of a hazard occurring, the likelihood of the hazard leading to an accident, and the severity of the worst possible potential loss resulting from such an accident.
>
> An *accident* is an event which occurs in an unsafe state and results in loss.

Whether explicitly or implicitly, a system is usually assessed at the outset with regard to safety considerations. Where it has been assessed that safety is an explicit requirement then we define (in a goal-oriented fashion) *safety-related systems* as those systems by which the overall safety of a process, machinery or equipment is assured [B. 89, Wic92]. As soon as the requirements are specified, they form the central plank for the development: the whole design process then becomes driven by the requirements (a fundamental view, which we adopt for the formal development). We refer to those requirements related to safety as *safety-related requirements* (or simply *safety requirements*).

The behaviour of systems can have consequences that vary in adversity, so we add another definition:

> A *safety-critical system* is a safety-related system in which the potential loss is very serious, a primary example being human fatality.

Many safety-related systems depend on computer software which monitors and controls various aspects such as physical equipment through direct interfaces. Computer

systems which act in this way as information processing components are known as *embedded computer systems*. Where software is a component of a safety-related system, we may refer to *safety-related software*.

Software is by itself relatively safe – it does not *per se* threaten great loss. However, software-related errors may be very hazardous and cause great risk. The software engineer's task as regards issues of safety may then be expressed as trying to ensure that the safety-related software contributes towards a safe overall system. The software engineer's brief may now be summarised as: to design and produce software with appropriate integrity and to provide assurance of this integrity *as part of the overall system.*

Safety is just one aspect that is desirable; a more general view of requirements may be encompassed in the framework of *dependability*. This is defined as "the trustworthiness of a computer system such that *reliance can justifiably be placed on the service* it delivers". A framework that discusses dependability, based on this definition, is given in [Lap93]. It provides, in particular, a classification of attributes that are typically needed for a system – availability, reliability, safety, confidentiality, etc. – and goes on to consider the factors that determine the extent one can place dependence on the system to satisfy these attributes.

The interdependence between the hardware, software and operating environment of the system motivates a discussion of approaches to systems safety, given next.

2.2 Traditional Systems Engineering Approaches

A safe system can be characterised as one in which risks from hazards have been minimised throughout system life. The process of providing hazard analyses and risk assessments are thus crucial activities to ensure the safety of the system.

2.2.1 Hazards Analysis

For any system, the provision of safety starts with hazard analysis, for which various techniques may be employed. Recognising that a system has many parts, one may, for instance, take a modular approach consisting of *System Hazard Analysis* (SHA) and *Subsystem Hazard Analysis* (SSHA). SHA studies the hazards associated with the system as a whole or the interfaces between its subsystem components, whilst SSHA studies how the operation of a given component affects the overall system.

These analyses are performed by applying a range of techniques which are well documented and generally covered by standards. They also range in scope: conducted at

the broadest level (with the greatest amount of brainstorming) are hazard and operability studies (HAZOP) [Che87]. HAZOP, devised originally to support the chemical process industry, takes a representation of a system and analyses how its operation may lead to an unsafe deviation from the intent of the system [Che87] with special attention to the environment of operation. It is a methodology that is now more widely embraced: guidelines are now being produced by the UK Ministry of Defence for systems with embedded programmable electronic systems [Def95].

Where a system is self contained, having its boundaries well defined, one focuses on the hazards that are internal to the system, which may be termed *faults*. Thus a fault is always a hazard, but not conversely (we do not, for instance, regard bad weather as a 'fault'). At this level, we have the techniques of fault tree analysis (FTA) [Bri83, VGRH81]. In BS5760 [Bri83] we have:

> "*fault tree analysis* ... consists of an analysis of possible causes starting at a system level and working down through the system, sub-system, equipment and component, identifying all possible causes."

In trying to determine possible causes of a fault in a fault tree, one can look at the operation of various components (which can be regarded as a HAZOP activity).

Other techniques include failure modes, effects and criticality analysis (FMEA and FMECA) [Bri91]. Further, as a project develops, one may perform design specific analyses such as design reviews, audits and walkthroughs. For a more detailed discussion of system hazard analyses with a software perspective, good coverage is provided in [Lev91].

2.2.2 Risk Assessment and Safety Integrity

In general, systems risk assessment is derived from data available – about hazards, analysed as above; from similar systems which have been implemented in the past; from the reliability assessments of components of the system being developed; and other sources. The result of risk assessment should be some kind of gradation and may be expressed in terms of what constitutes *tolerable* and *intolerable* risk. Tests applied for regulating industrial risks echo those we take ourselves in our personal lives and involve determining whether risk is unacceptable, acceptable or somewhere in between. There are a lot of factors which determine what is 'tolerable' or otherwise, so both quantitative and qualitative analyses are used, e.g.s graphs and classifications [BR93]. Using a risk classification of accidents according to frequency and severity usefully serves as a relatively simple basis for its determination. In safety-critical systems we focus naturally on the resulting 'intolerable' risks or those risks which are close to intolerable.

Only when the risks have been assessed can we decide upon the necessary levels of safety that we expect the system to achieve from its various functions. This is the issue of safety integrity, which is defined as:

> *Safety integrity* is the likelihood of a safety-related system achieving the required safety functions under all the stated conditions within a stated period of time [Wic92].

Therefore, the task is to deliver a system of sufficiently high integrity to meet all the requirements. As the production of a system is a process, these procedures have to be maintained throughout the development, requiring ongoing hazard and risk analyses both in terms of the envisaged goal, the 'end product', and in terms of what may be regarded as an evolving design. As more information is revealed about possible operating conditions, systems safety analysis is augmented. Consequently the activities contributing to the integrity may be characterised by two kinds of requirements:

1. generation of new system safety requirements resulting from the design and development of the system
2. ensuring that what is being built meets the requirements that have already been specified

The first of these is requirements analysis and consists of activities mentioned already, including hazard analysis.

The second of these are *reliability engineering* techniques, whose consideration may have to be sustained throughout the development as the design evolves with modification to interfaces, rearrangement of components or other kinds of changes. The main measures to achieve reliability emerge as a result of employing the techniques mentioned – such as FMECA, and FTA – and consist of fault forecasting, fault removal, fault avoidance and fault tolerance, together with methods that verify that the design has achieved the integrity required. They may be regarded as a combination of forward-looking and backward-looking techniques, operating on a model of cause and effect:-

1. "If we start from here, where will this lead us? Where will there be a failure or failures?"
2. "What faults might we expect? How may they be arrived at?"

In general, FMEA and FMECA deal with the former and FTA with the latter.

In this approach, ensuring safety may then be characterised simply as a process of reducing risks to appropriate effect. This consideration depends upon resources available. If a risk falls in between the states of 'intolerable' and 'acceptable' then any risk must be reduced to 'as low as reasonably practicable'. This is the ALARP principle as illustrated in Figure 2.1. The width of the triangle is proportionate to the level of risk and thus also to the amount of resources that can be justified to reduce it.

In summary, a system is conceived to perform certain functionality in the face of hazards, whose risks are determined. These contribute in turn to further safety-related requirements that are derived for the system and for which safety integrity levels are stipulated. Achieving the integrity means in effect that risks must be reduced, for which we may apply the ALARP principle.

A comprehensive survey of risks and safety integrity is provided in [BR93].

2.2.3 Safety Integrity and Assurance

Finally, we must answer the question, "What assurance can you provide that this system is of such integrity?" In order that a system may be certified as safe, there must be provided a document which details the justification of its safety. This is called the *safety case*. It contains a record of all hazards, known as a *hazard log*, and various arguments indicating why the system will reach the required safety levels in the face of the stated hazards. The safety case brings in all the aforementioned risk analyses, risk reductions and other integrity and reliability measures, often presenting various statistical evidence.

This is a considerable task which involves lots of documentation. Accordingly, software has been developed and used to support this process, one tool being SAM (*Safety Arguments Manager*) that is able to support the process of developing safety cases, with the intention of uniting both formal and informal material (including the often important assumptions made) [FHMS93]. Here, software plays an effective supporting role, important, though not as crucial as many of its more direct applications.

2.2.4 Software Safety

Software may be engineered within the above kind of system setting and is a system in itself, but there are some striking distinctive qualities which software possesses. Regarding its theoretical foundations, the discipline of software engineering is underpinned not by classical laws of mechanics or thermodynamics, but by somewhat more abstract discrete mathematics. Consider for instance that a program can in theory always be relied

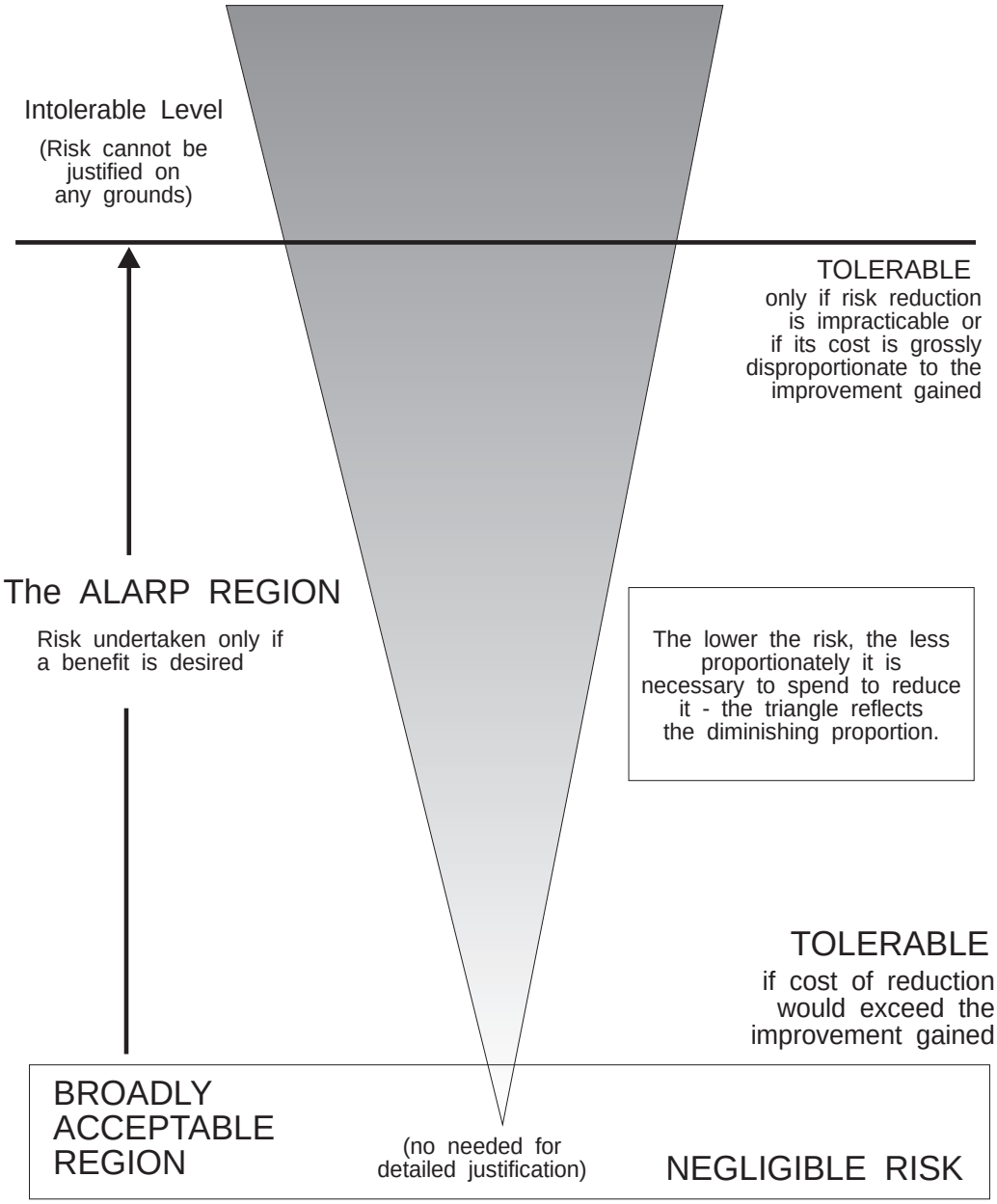

Figure 2.1: ALARP model of risk levels

upon to carry out its instructions, whereas in contrast a piece of hardware is subject to wear and tear and will eventually fail.

Software elements often have very dynamic relationships with diverse areas of the rest of the system, the structures being much more fluid than in hardware. Hence, it cannot easily be assumed that a piece of software can easily be produced to 'slot in' as some kind of simple 'fix'. A new external system requirement may need an unexpectedly complicated software reconfiguration.

All of this amounts to what was coined in 1968 'The Software Crisis' at a meeting organised by NATO in which were convened some 50 top computing professionals. This crisis had as its root cause the problem of complexity brought about in many cases by sheer length of programs combined with a poor control over how each line of code affects the overall system. Almost three decades later, this problem still remains as a recent review indicates [Gib94], and for safety critical systems, the problem is arguably still more keenly felt [McD93].

In order to tackle the complexity issue, there have been initiatives at several levels, some of which are on a major industrial scale. The United States, which handles most of the largest software projects, has produced a number of initiatives aimed at putting software production on a sound commercial footing. The Software Engineering Institute, funded by the military, has produced a Capability Maturity Model (CMM)[PWCC95] by which may be assessed the quality of management in a software engineering team. Some companies have responded to this and subsequently seen their productivity increase significantly. The NIST has recently created an Advanced Technology Program to encourage a market in component-based software.

There has been a lot of effort devoted to the management of software and the drive to make this relatively new discipline line up with the rest of industrial engineering; figures indicate that the software has become more reliable since there are fewer bugs per N lines of code. However, how can one relate this to requirements for operation that are specified in terms such as 'failure rate of less than 1 in 10^9 hours of service'? These are requirements in response to pretty well intolerable risks and at first sight this presents a frighteningly tall order; perhaps only now is the enormity of the task clear.

Statistical quantification of software reliability *in actual operation* is problematic: besides the fact that software simply hasn't been around long enough for statistics that yield sufficiently high reliability, its worth noting that each piece of software is individually tailored, unique to a much greater extent than hardware set-ups. It has been argued (in [BF93]) that probabilistic and statistical methods are inadequate anyway, though this claim

has been countered elsewhere.

As a prerequisite to a realistic appraisal of the integrity of an embedded system, there needs to be an understanding of the nature of the effects of software on risk and integrity, and hence ways of ascribing suitable figures. It would then become clearer what would constitute appropriate justification. These important issues still lie open, which we touch upon in the next chapter; a proper appraisal will take many years and require considerable experience from and cooperation between academic and industry. There are fortunately some working groups that are moving towards its fulfilment and which have been active over a few years. These include the UK DTI/EPSRC Working Group which has examined over 30 projects in the UK [UK 97], and PROCOS, which is a European group supported by ESPRIT.

Once these issues are better understood, we will know more about the extent of the task of providing the high integrity. It may be that this objective will only be achieved when it is accepted that this particular industrial sector is more inextricably and subtly dependent than others upon the underlying mathematical theory. Evidence that this is recognised may be gleaned from industrial initiatives such as the 'cleanroom approach' to programming, as being experimented with by IBM and which incorporates formal notations such as Z[Spi92]. Thus the case for formal methods arises, detailed later.

2.3 A Generic Framework: The Safety Lifecycle Model

In recognition of the distinctive nature of safety-related systems, there has been developed what is now the widely accepted Safety Lifecycle Model, which is an extension of the standard Waterfall Model in engineering [B. 89, Wic92, BR93]. It is generic in that it is valid across industrial sectors. The model is depicted in Figure 2.2.

In order to produce safety-related software according to this framework, various techniques are recommended. These include the application of structured analysis techniques to generate a visible modular construction (the principles of modularity are expounded in [Par72]), and diversity in design, implementation and maintenance to avoid faults due to common mode failures. Many such techniques are very widely applicable, and although they are usefully brought into the safety-critical context, there is not so much literature devoted solely to their use in this specific area. Nevertheless, material is available: for instance, there have been reviews such as [CGW91] to help designers and managers as to the suitability of mainstream programming languages for safety-critical systems.

A set of system-wide guidelines may not shed enough light on what actually con-

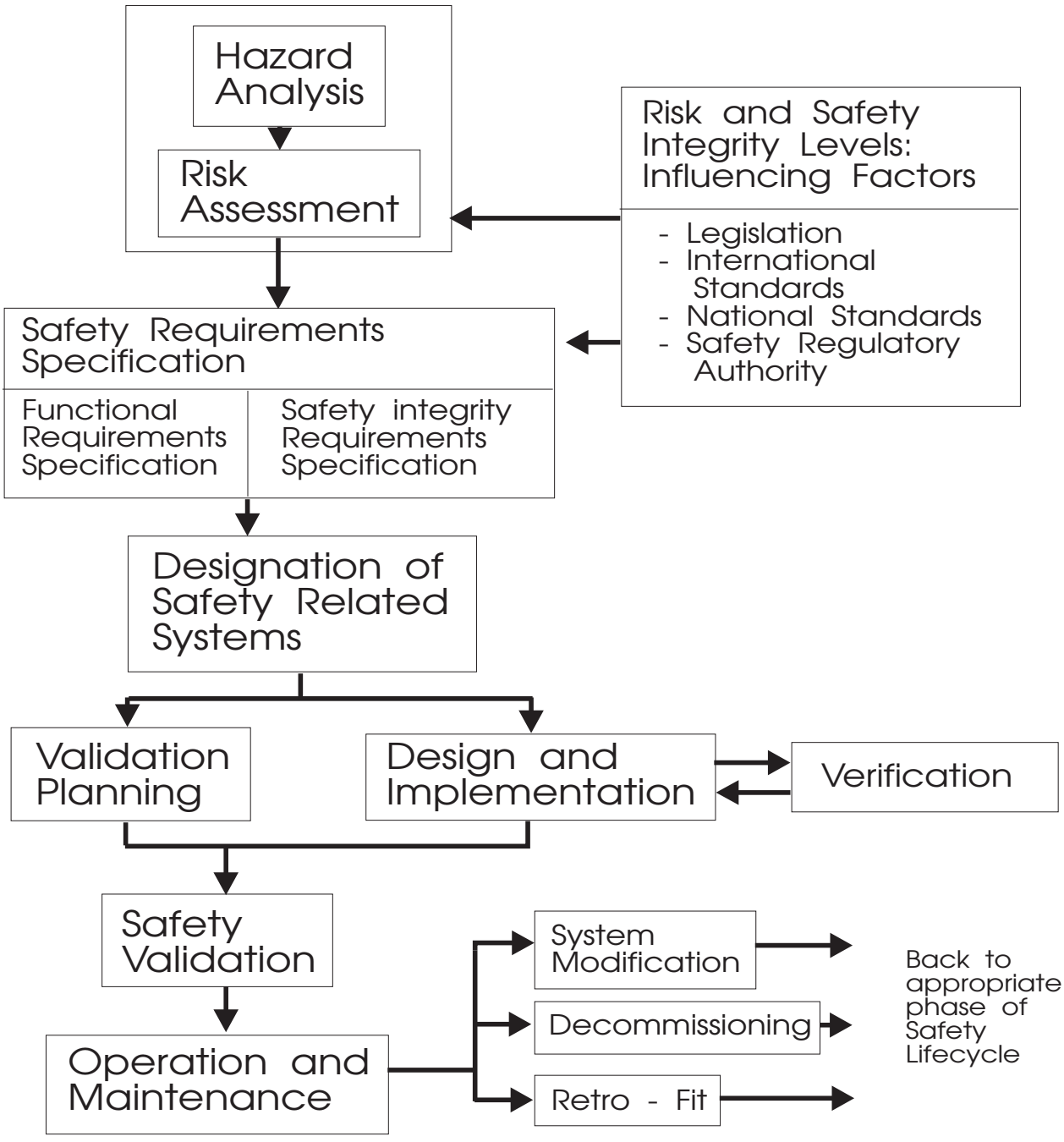

Figure 2.2: Safety Lifecycle Model

stitute the safety issues unless they are unified under a specific focus. Safety requires a lot of integrity and this is recognised in the safety lifecycle model which separates specification of safety requirements into purely *functional requirements* (what the system should do) and *safety integrity requirements* (the level of dependability expected of these functions). The safety integrity requirements are calculated individually for each of the functions previously identified. Having done this, one may concentrate on providing high levels of assurance on the safety-critical aspects. (As an aside, note that if it is the case that there is always some functionality needed to ensure safety then there is an openly recursive loop.)

We intend using the safety lifecycle model as a basis, with a view to ascertaining its suitability to support the production of formal models with high integrity. Our contention is that provided we treat carefully the *non-functional requirements* and put forward a selection of viewpoints and methods highlighting further the safety concepts, which are often subtle, then the lifecycle model can be effective.

Software-related systems in general may be characterised broadly by two fundamental notions – *data* and *behaviour*. Associated with the former are tasks of information processing, whereas with the later there is action and response related typically to the input/output of such data. Those systems, which are extensively involved in responding to their environment are called *reactive systems*. They are "systems that are heavily **control driven** or **event driven** ... Their role in life is ... predominantly to react to many different kinds of events, signals, and conditions in intricate ways. Reactive systems need not necessarily be concurrent, but usually are." [Har87]. Such systems include communication systems, for which the Flexport communications protocol studied in this thesis is an example.

2.3.1 Safety Aspects in Software for Medical systems

In this section we illustrate how safety aspects may be highlighted when considering the use of software within a medical setting. One technique is to provide alternative views on the same system, depending upon the task at hand.

For the task of requirements analysis, one view can consist of analysing a system in terms of *scale* – the degree of granularity; and level of *abstraction* – the extent to which selected details are ignored. As an example, let us take as a specific scenario a Medical Information System in a hospital, which allows various patient details to pass through a network between, for example, the central file server, the operating theatre and the intensive care unit (ICU). Such a system has numerous interdependent software components at various levels, ranging from Windows applications to 'low level' network drivers. An ICU, a

component, has its own information system spanning a network of medical devices attached to bedsides. It consists of a number of component computer systems managing various tasks such as information processing of patient lifesigns – a Patient Data Management System (PDMS) – and the supporting communications network which should guarantee the correct delivery of the data from a bedside to the nurse's console.

Thus we may derive a component view of the system, an approach similar to SHA and SSHA, concerned with:

1. Internal-External *interaction* between system and environment, i.e. the interface.

2. Internal Localised Safety properties – workings hidden from the external environment.

The terms 'internal' and 'external' are relative: through successive abstraction we may choose that on one level we identify a system as a component co-operating within an environment and on another level, where it is of no relevance in the chosen setting, as hidden or swallowed up. A nurse or doctor does not wish to know about the sequence of bits which travel along the cables between the bedside communications controller and a device communications controller, say. However, their diagnoses will certainly depend upon the correct transmission of the 0s and 1s. The software which is used by the medical staff has *abstracted out* from the internal workings of the system, workings which are internalised safety properties, as in 2.

A nurse will be keen, however, to know about what is displayed on the console: this involves safety aspects in interaction, as in 1. We make the extra distinction between the human-computer interaction (HCI) and any computer-computer interfaces, for the field of HCI deserves a great deal of consideration. A typical fault tree showing paths to failures or errors in a Patient Monitoring System will usually contain a significant proportion of failure modes (root causes) which involve some inappropriate response to a computer output. It could be that a nurse or a doctor does not interpret correctly a certain visual display or an audio signal. There is always a chance of misinterpretation, but more research and consultation between designers and users would minimise this risk. Fortunately, HCI has made great strides in the past decade as can be seen in a textbook such as [J. 94].

For the task of achieving the integrity levels stipulated for certain componets in operation, one view can focus on the nature of the faults. Requirements can then be refined to accommodate these faults. This can be considered as part of FMECA. Using the same medical example, we may classify the nature of potential faults (with regard to examining their effects) as:

1. *Explicit global* Consider instances where an oxygen mask slips off a patient or where the plug is accidentally pulled out of a socket. Here we deal with safety properties which are expressed in terms of the overall system as viewed externally: these are in terms of the "physics" of the environment give rise to explicit requirements. We refer to the external system as the *plant*.

   By their external nature, particularly if they are relatively large scale considerations (as with nuclear power stations), government legislation may be suitable for setting appropriate levels.

2. *Implicit local* Consider instances where a system simply freezes up or engages in an endless loop state where no further useful action is available. The safety properties to prevent these occurrences may be ignorant of the overall application and give rise to requirements in terms of the internal system, which we refer to as the *controller*. They are often obscured and potentially more dangerous since they can unexpectedly contribute to dramatic external accidents. Distributed systems, where there is a confluence of more than one stream of data, may be particularly prone to such faults. The preparation of a dinner in a hotel kitchen is a distributed system consisting of a number of cooks, each attending to individual dishes: they exercise their own quintessential culinary finesse, but have to co-operate towards a common gastronomic goal, sharing the sinks, the cooking utensils etc ... its smooth operation requires a collective understanding of each individual's role.

Applications in the medical sector are numerous: an indication of the extent of the coverage is given in a special issue of *IEEE Computing and control* [Fel95] in which there are articles on Computerised conformal radiation therapy; a medical imaging system in diagnostic microscopy; a communication system for wheelchair-mounted medical robots; medical robotics itself; and programmable electronic medical systems. Other applications include medical administration – database systems of various sizes, and amongst the more ambitious projects are those concerned with the application of virtual reality for remote surgery.

2.3.2 Observations on software within the System setting

In summary, in the safety-related system as a whole, the methods employed start with hazard identification and proceed through to risk assessments. Then the required safety integrity may be determined and carried out together with subsequent assurance of this integrity. In systems which have safety-related software and especially safety-critical

software, this assurance is difficult, almost impossible, to provide as we've already indicated, though it can be revealed when a system has insufficient integrity.

In order that the design process may fulfil all the requirements to the appropriate safety integrity level, a suitable combination of scientific, engineering and management practices may be applied. For instance, management can ensure that scientists and engineers work harmoniously towards a practical goal, within the current social, economic and political climate. Most of these will also be needed to assure standards authorities that the system may be certified as safe. A safety lifecycle model provides an appropriate coherent focus in which to make this a more realistic proposition.

2.4 Formal Methods for Safety-critical systems

In this section we show the need for the use of formal methods, providing some informal definitions of the main concepts.

2.4.1 Motivation for their use

Providing high integrity systems with embedded software requires a careful argument for its justification. Demonstrating such exacting requirements through sufficient statistical evidence based on testing and other general reliability measures has been shown to be doubtful. Thus, some other kinds of arguments have to be written, which must be precise – in language that is well-defined, whose meaning is clear, and with the ability to prove statements without doubt. Since natural language is unable to fulfil such demands, the only possible solution is to use a mathematical approach – formal methods.

A formal approach is ideal for *verification*, the activity guaranteeing correctness, i.e. (to paraphrase) that we are *building the system right* and particularly that successive refinements of a specification are consistent with each other. More than that, the discipline which they encourage often leads to a more careful analysis of the most basic assumptions and definitions in the design, a benefit which is often understated. In particular, they may point to ambiguities in the requirements definition. Formal methods is thus effective for *validation* – making sure that we are *building the right system.*

Unfortunately they also have their costs, but the obstacles to their use are being overcome: publications such as [BS93b, Tho93a, HB95] indicate the progress that has been made. As these formal procedures have only been applied recently, at the moment their deployment needs to be carefully targeted. The case for formal methods increases as stringency on the failure rate increases. A 'modest' failure rate may be achieved by standard

software engineering, where techniques such as product testing and case analysis may suffice, whereas for stricter failure rates, such analysis is insufficient and perhaps only formal methods will do. They are particularly suited to requirements analysis where the stakes are exceedingly high in terms of the development of reliable safety-critical software [Bye92].

2.4.2 Definition

Formal methods may be defined as a branch of discrete mathematics which deals with the logical analysis of forms and their semantics (meaning), with a specific application domain being computing. They usually consist of two parts:

1. a *formal calculus* (or *formal system*) which is a symbolic system in which are defined axioms, having some denotation as formulae; a precise *syntax* that defines how the axioms may be put together; and relations that enable the *deduction* of properties purely as the conclusion of arguments that are valid through the system's syntax.

2. a *formal language* is a formal calculus which has also an interpretation of the formulae – *semantics*. Further rules constrain what constitute valid (meaningful) formulae and the properties that may be deduced. A given calculus may have infinitely many interpretations.

The fruits of the first part are propositions and theorems which express properties about the formulae. The second part generates the same kind of results, but also allows the more liberal view that formal methods may be regarded as a mathematical approach to *reason about* any system, be it an industrial factory or an abstract machine. The mathematical disciplines used are based on set theory, predicate logic and algebra; the 'methods' in formal methods are techniques related to these disciplines.

This is rather a contrast with the usual conotation of 'methods' in industry, which are procedures to generate working products. As many authors have pointed out, the general lack of such an industrial view, has proved a great stumbling block [Bri92]. This provides our motivation for examining procedures to support the formal approach for integration into a working methodology. As a start, we conceive the part played by formal methods in software development as illustrated in Figure 2.3, which fulfils the two kinds of requirements expressed in section 2.2.2. First the envisaged system is conceptualised in terms of broad requirements, whence we may translate it into some formal unambiguous representation – an abstract model. We then employ mathematics to analyse and reason about our system through the model, establishing the model's validity.

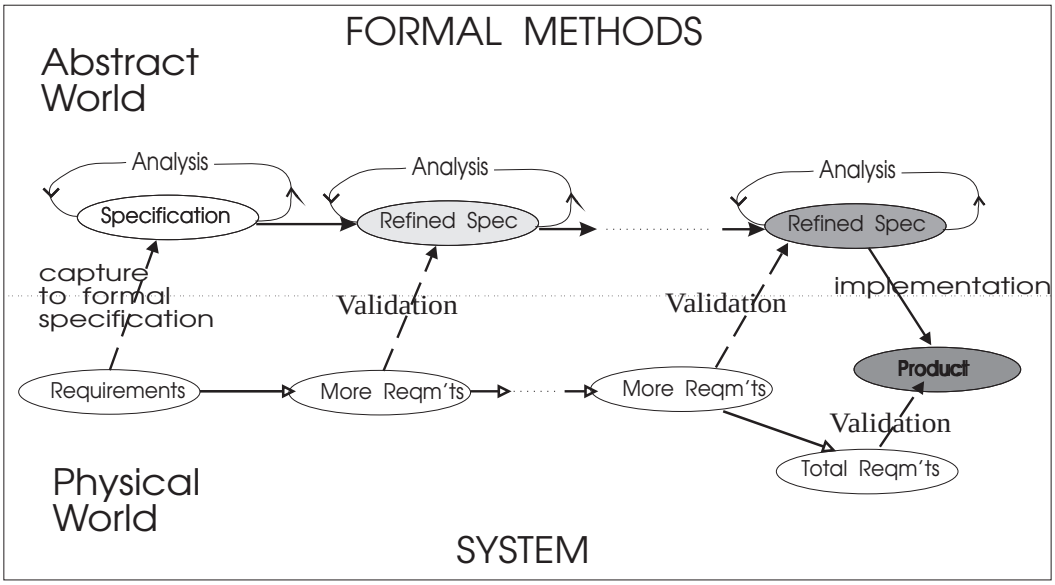

Figure 2.3: Formal Methods: capture, analysis and refinement in Software Development

When this is achieved satisfactorily, the subsequent development consists of successively refining the model towards physical implementation in such a way that it is consistent with the chosen formal notions of refinement. It is necessary to verify that the steps are made in a manner that preserves the required functionality and properties according to these notions. In Figure 2.3, verification is performed in the **Abstract World** and is denoted explicitly by *Analysis* (for checking internal consistency) and implicitly in the rightward arrows (for checking consistency of successive specifications).

The traditional role of validation in engineering is, typically, to check that a product or implementation meets its requirements. This may be extended in the software setting by taking the view that validation is the process of showing that we are build-*ing* the right system. An incremental view is particularly pertinent in the formal setting where mathematical relations relate initial incomplete or abstract models with subsequent complete 'implementations'. To clarify, we can use the term 'model validation' for a model that is being refined toward a product, and when this is understood, we talk simply of 'validation'. Figure 2.3 reflects this view.

Thus validation may be considered as a repeated process of interpreting requirements in the **Physical World**, capturing this formally (indicated by the upward arrows) and then checking and analysing (in the **Abstract World**). The process continues for successively more requirements as the design possesses larger scale and/or finer details. The model may thus be seen to evolve in the abstract setting, dependent upon the real-world

setting. Working in this fashion should lead smoothly towards a finished product.

For this to be well grounded, regarding the verification we require *a priori* that the formal system itself is *sound* (or *consistent*), i.e. that it does not enable the deduction of a contradiction from the axioms. For the validation, we require that the formal system enables that any model that is valid to be demonstrated as such. This is the property of *completeness*: a formal system is *complete* if every (semantically) valid formula can be proved (syntactically) from the axioms. Sound and complete formal systems are kinds of assurance in that they guarantee proofs, at least in theory. If proofs are conducted in systems which have not been shown to have these properties, then some additional justification would be required.

Alternatively, the process of refinement may be considered in terms of successively reducing the size of a set of valid implementations: as the requirements increase, the family of valid systems decreases.

2.4.3 Formal analogues of safety-related aspects

Tackling safety from a general engineering perspective led us to considering hazards and risks. In our mathematical treatment, the corresponding conditions that have an important bearing on safety are expressed in terms of certain kinds of properties. Mapping safety engineering concepts to the formal domain raises a number of issues, especially that of expressiveness of the respective formalism, discussed in the next chapter. Here, we give only an indication of their potential.

We start off by citing some of the formal meanings of some safety-related properties. Below is a simple classification into two kinds with respect to temporality – *safety* and *liveness*, properties that were originally defined for multiprocess programs in [Lam77]. We quote the definition of [BA90]) and then give some typical examples:

- **safety property**: The property must *always* be true.
- **liveness property**: The property must *eventually* be true.

The first definition is really an invariant, and does not convey the sense of protection from loss or injury. The most common example of a safety property which conveys a more familiar meaning is "something 'bad' never happens", where 'bad' has a formal interpretation, reflecting some common sense view of hazards or accidents. Things "happen" when certain actions or events occur. Thus, one safety property may be expressed as: there

is always absence of *deadlock* (the state where no actions are possible), which may be regarded as a kind of service omission. More subtle is absence of *livelock* (the state where no useful actions are possible) which may also be regarded as a service omission. An important instance of livelock is *divergence* in which a component gets caught up in an infinite internal loop.

One example of liveness is *fairness* – some event(s) must occur eventually, perhaps precisely when certain other processes have reached certain states, or, more loosely, on being available infinitely often. Such liveness may also be expressed by "there is always some action or event which the system may perform to usefully evolve."

Many safety requirements make reference to time: for example, a hazard would be created if an oxygen mask slipped off a patient in an ICU, in which case the crucial requirement is that a nurse will be alerted *in time* to replace it. The issue of handling time in a formal manner is a philosophical problem, for which a prime concern is the question: at what stage in development should time be mentioned explicitly? In our example, it is of paramount importance that the network has integrity, i.e. that the correct sequence of actions is carried out with sufficient urgency. In practice, a well designed network will correctly transmit an alarm signal in a split-second. The real difficulty may lie in ensuring that the signal actually gets transmitted and that it doesn't get thwarted by livelock. On the other hand, the implementation may well perform actions to synchronise with the ticking of a global clock.

An overview of most of the formalisms that have been used for the analysis of safety-critical systems with real-time aspects has been given in [Ost92]. The survey provides a broad classification into three main directions, reflecting successively greater formality: the first is a brief look at real-time programming languages such as ADA and OCCAM; the second considers structured methods and graphical languages such as STATECHARTS; and the third, the largest, examines logics and algebras. It is by no means complete, but a useful reference source nevertheless.

2.5 Introduction to main formalisms used in thesis

In this section we introduce some formal notations to illustrate their ability to model behaviour and capture certain properties, particularly safety and livenesss. In view of our Flexport case study, we do this for concurrent systems and employ a *dual language* framework of process algebras for developing system models and temporal logic for specifying requirements.

2.5.1 Transition Systems and Labelled Transition Systems

Transition systems are especially suitable for modelling the operational semantics of systems (both plant and controller). They are ideal as a basis for the logical analysis for safety-critical properties – the modal and temporal properties to be treated later. We introduce some notation and definitions, generally following [Sti92b] to illustrate the type of reasoning we can perform.

Definition A *transition system* is a pair $\mathcal{T}$ consisting of a non-empty set $\mathcal{S}$ and a non-empty set $\mathcal{R}$ of ordered pairs (s_i, s_j) with s_i and s_j both in $\mathcal{S}$.

$\mathcal{S}$ is a set of states, and $\mathcal{R}$ is a set of *transitions* from $\mathcal{S}$ to $\mathcal{S}$. To model systems, and in particular their internal actions, we require more information to be specified.

Suppose we wish $\mathcal{S}$ to represent, for instance, the possible locations and configurations of a robot on an assembly line. This system is dynamic and has transitions from one state to another – denoted by the set $\mathcal{R}$. We make explicit the fact that such transitions are due to *actions* and introduce $\mathcal{L}$, a set of *labels* to denote the set of all actions possible in this system (perhaps containing elements such as 'move left', 'rotate right arm 90^o right', ... etc).

It is now useful to classify the transitions in terms of actions, so we define two types of *relation* (a set of ordered n-tuples) – here an ordered pair, and an ordered triple. ($\times$ denotes Cartesian Product).

(i) A subset of $\mathcal{S} \times \mathcal{S}$, denoted by $\stackrel{a}{\rightarrow}$, representing those transitions possible for a certain action $a \in \mathcal{L}$.

(ii) A subset of $\mathcal{S} \times \mathcal{L} \times \mathcal{S}$, denoted by $\rightarrow$, consisting of all possible transitions for the labelled set $\mathcal{L}$.

We now define a *labelled transition system* using the relation defined in (i).

Definition A *labelled transition system* is a pair $\mathcal{T} = (\mathcal{S}, \{\stackrel{a}{\rightarrow} | \ a \in \mathcal{L}\})$, where $\mathcal{S}$ is a non-empty set (of states), $\mathcal{L}$ is a non-empty set of labels, and for each $a \in \mathcal{L}, \stackrel{a}{\rightarrow} \subseteq \mathcal{S} \times \mathcal{S}$.

We write $s \stackrel{a}{\rightarrow} s'$ if either $(s, s') \in \stackrel{a}{\rightarrow}$ (or if $(s, a, s') \in \rightarrow$).

The definition leaves open how the structure of S may be represented. They allow easily for parallel systems to be interpreted on them.

2.5.2 Process Algebras

Transition systems given as a list offer few clues as to structure and it is difficult to follow the flow of transitions. In view of this, languages have been developed to provide these, thereby facilitating reasoning. They include approaches based on *Petri Nets* [Pet81], one of the first formalisms to deal with concurrency. One family of languages that has evolved with various influences (including Petri nets) and which has proved very useful is *process algebra* (or *process calculi*). Examples of process algebra include ACP[BK84, BW90], CSP[Hoa85], and CCS[Mil89].

Process algebras are particularly good at expressing the structure of concurrent systems in terms of simple building blocks – labels and structural combinators (various operators). They are able especially to model systems at different descriptive levels. The operational meaning (or *transitional semantics*) of systems is in terms of observable behaviour on transition systems.

2.5.3 LOTOS

In this thesis the language used is LOTOS (Language Of Temporal Ordering Specification) [ISO89a], an ISO standard formal language which was developed principally for the specification of OSI protocols and services – ISO 8807. It has two parts – a process algebra, referred to as Basic LOTOS, derived largely from CCS and CSP [Hoa85] and a *data type language*, based on ACT ONE [EM85], to structure the behaviour in terms of the data which passes through system. This rich combination is suitable for distributed systems in general and particularly good at expressing behaviour of communications protocols, so it is a natural choice here. A good tutorial is provided in [BB89].

We do not define LOTOS here – please refer to Appendix A, where the transition system with rules is given for LOTOS.

2.5.3.1 Examples

We provide some simple example specifications here which will be used later to illustrate some validation.

Suppose we wish to model a householder (Laurel) replacing old gas cannisters, now empty, with new ones. He has his old cannisters on the doorstep, which he can pick up and then deposit in the back of the lorry belonging to Hardy, the Gas man. Likewise, Laurel can pick up new cannisters from the back of the lorry and deposit them on his doorstep. The cannisters are heavy, so that once one is picked up, it must be deposited before another

one can be moved. We represent Laurel's 4 simple actions of picking up and depositing cannisters (which proceeds *ad infinitum*) in the following process:

```
process Replace[collect_old, deposit_old, collect_new, deposit_new] : noexit:=
  collect_old; deposit_old; Replace[collect_old, deposit_old, collect_new, deposit_new]
 []
  collect_new; deposit_new; Replace[collect_old, deposit_old, collect_new, deposit_new]
endproc (* Replace *)
```

Suppose Hardy decides to get out of his cab and lend assistance. He agrees to collect the old cannisters off Laurel, put them into the back of his lorry and also to collect new ones and pass them to him.

We now model their overall behaviour as two concurrent processes which must synchronise their actions in the handover of the cannisters. Their behaviour is identical, so we define one process, 'Transfer' which represents the behaviour of receiving from one side and passing to the other:

```
process Transfer[rec_l, send_r, rec_r, send_l] : noexit :=
  rec_l; send_r; Transfer[rec_l, send_r, rec_r, send_l]
 []
  rec_r; send_l; Transfer[rec_l, send_r, rec_r, send_l]
endproc (* Transfer *)
```

We then synchronise their individual actions – on Laurel's depositing of old cans and Hardy's passing on of new ones – and use process instantiation. This behaviour, denoted by GAS, say, is represented in the following LOTOS expression:

```
Transfer[Laurel_collects_old, deposit_old, collect_new, Laurel_deposits_new]
 |[deposit_old,collect_new]|
Transfer[deposit_old, Hardy_deposits_old, Hardy_collects_new, collect_new]
```

This can then be considered as a prototype design for the transfer of gas cannisters, but does it work reliably ...? In the next section we analyse the system drawing in safety-related aspects to perform validation using logical analysis.

2.5.4 Modal and Temporal Logics

Modal and temporal logics are structural languages dealing with propositions of formulae at given states, interpreted on transition systems. A large body of theory has

been built up for these languages, dealing e.g. with soundness and completeness, enabling expressive reasoning about properties of transition systems in general. Applying this theory can reveal information about particular transition systems.

Modal logics focus on actions which provoke change, and represent well local behaviour – good for modelling properties such as capability.

Temporal logics focus on resulting states, and the ongoing nature of systems, suitable for global behaviour – good for modelling properties of overall liveness and safety.

In the next subsections, we introduce modal and temporal logics in order to express some safety requirements, specifically for the 'Gas cans' example. A detailed guide to this subject is provided in [Sti92a].

2.5.4.1 Modal Logics

Modal logic provides a system for discriminating between processes in terms of local capabilities. Formulae are used as the basis for the testing and are assigned a *valuation* for each state. So, typically, if we know whether or not a formula holds at a given state, then we assign a valuation of either true (`tt`) or false (`ff`).

The formulae are built up inductively according to an abstract syntax definition, by some union of formulae and prefixing using members of a set of labels - such as $[a]$ ("box a") and <a> ("diamond a"). The following is taken from [Sti92b], itself being a slight generalisation of Hennessy-Milner logic [HM85].

$$\phi := \texttt{tt} \mid \texttt{ff} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \ \phi_2 \mid [K]\phi \mid \ <\mathrm{K}> \phi$$

Here, K denotes a set of labels. For a set containing a single element, $\{e\}$, we write just e.

Thus a formula is either *true* (`tt`); *false* (`ff`); a conjunction ("and") of formulae $\phi_1 \wedge \phi_2$; a disjunction ("or") of formulae $\phi_1 \vee \phi_2$; or a *modalised* formula $[K]\phi$, <K> ϕ.

2.5.4.2 Modal Logic for Processes

In a process algebra, we've seen that processes evolve through actions according to transition rules: recall $g; P' \xrightarrow{g} P'$ is the rule that allows a process to evolve to behaving like process P' after a transition g. We can relate modal logic to processes by regarding the labels as actions and the modal formulae as interpreted on states of a given process P, which we henceforth identify with behaviour expressions. Each action takes us to a new

process. We relate processes to each other through definitional equality: $P \stackrel{def}{=} g; P'$ means that P has initial action g and then behaves as P'.

To relate formulae to processes, we can define a valuation for formulae on states: we denote by $P \models \phi$ the property that P has (or satisfies) ϕ. The satisfaction relation $\models$ between formulae and processes is done inductively on the structure of the formulae:

$P \models \mathtt{tt}$		
$P \not\models \mathtt{ff}$		
$P \models \phi_1 \wedge \phi_2$	iff	$P \models \phi_1$ and $P \models \phi_2$
$P \models \phi_1 \vee \phi_2$	iff	$P \models \phi_1$ or $P \models \phi_2$
$P \models [K]\phi$	iff	$\forall Q \in \{P' \mid P \stackrel{a}{\rightarrow} P'$ and $a \in K\}.\ Q \models \phi$
$P \models <\mathrm{K}> \phi$	iff	$\exists Q \in \{P' \mid P \stackrel{a}{\rightarrow} P'$ and $a \in K\}.\ Q \models \phi$

Interpretation

Every process has (axiomatically) the property *true* (`tt`), whereas no process has the property *false* (`ff`).

A process has the property $\phi_1 \wedge \phi_2$ if and only if ('iff') it has both properties ϕ_1 and ϕ_2; it satisfies $\phi_1 \vee \phi_2$ if and only if it satisfies (at least) one of these components.

We assign meanings of the modalised formulae, with respect to the transitional behaviour of processes. The modal operators '[ ]' and '<>' express necessity and capability. Thus P has the property $[K]\phi$ if after every performance of any action in K each resultant process has ϕ; $<\mathrm{K}> \phi$ has the property that there is some event in K such that the resultant process satisfies ϕ.

The two inductive definitions above allow us to express local behaviour of processes using modal formulae.

Some Properties for Gas cans example

Let $\mathcal{A}$ denote the set of actions available to a process, and let $-K$ be the set consisting of the set $\mathcal{A}$ less the actions in K. Then for GAS, we have the following formalisation of a safety property:

- *absence of deadlock* may be expressed by $GAS \models < \mathcal{A} > \mathtt{tt}$.

  (On the other hand, *deadlock* may be expressed by $GAS \models [\mathcal{A}]\mathtt{ff}$).

We can express some simple liveness properties, stipulating sequences of actions that should be possible. The initial behaviour R, say, of the process 'replace', having the capability of collecting either the old or new cannisters, is expressed by:

$$R \models < collect\_old, collect\_new > \texttt{tt}$$

The necessity that immediately after collecting an old cannister, it must then be deposited is expressed by:

$$R \models [collect\_old](< deposit\_old > \texttt{tt} \wedge [-deposit\_old]\texttt{ff})$$

Some approaches to and issues concerning the validation of these properties for a particular process definition are treated in a later section.

2.5.4.3 Temporal Logics

Temporal logic provides a system for demonstrating various safety properties. There are many temporal logics, in each of which safety, liveness etc. may be characterised. A recent paper that illustrates the formalisation of many such properties is [Sis94]. In our case, to be consistent, a temporal logic is formed by extending the modal logic above by the introduction of just one type of operator which captures appropriately the need for persistence of modal properties. This is the fixed-point operator.

Definitional equality ($\stackrel{def}{=}$) can be used to describe properties as well as processes. A process definition for an endlessly dripping tap might look like:

$T \stackrel{def}{=} splash; T.$

The property of persistently being able to 'splash', as satisfied by this process, may be defined by a temporal equation:

$Z \stackrel{def}{=} < splash > Z.$

Here Z is a *propositional variable*, which can have as values a number of solutions, each expressing various properties, though all saying something about the capability of 'splash'ing. This is a *recursive equation* - the Z in the right hand side has also, by definition, solutions satisfied by $< splash > Z$ and so on ...

Such equations can be valid for a whole host of different process definitions. Solutions to them are sets called *fixed points*. It may be shown that there exists both a least set μZ and a largest set νZ of fixed points: it is these two sets which are especially useful in the modelling of ongoing safety-related properties.

This extra concept is added to the modal logic to produce a *modal μ calculus* defined inductively by:

$$\phi := \texttt{tt} \mid \texttt{ff} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid [K]\phi \mid <\mathrm{K}> \phi \mid \nu Z.\phi \mid \mu Z.\phi$$

Safety can now be expressed as a persistent property that some bad state is never reached. Suppose $\neg\phi$ is only true at bad states then safety is expressed as:

$$\nu Z.\phi \wedge [\mathcal{A}]Z$$

i.e., the current state is good (ϕ holds) and any action will lead to the same equation – the Z in the right handside is a *bound variable*.

All the modal properties above, which held in the local setting, can now be made ongoing using the fixed point operators. Hence we can apply this to our parallel system, GAS, requiring as a safety property that there is ongoing absence of deadlock:

$$GAS \models \nu Z. < \mathcal{A} > \mathtt{tt} \wedge [\mathcal{A}]Z$$

A liveness property we seek is that eventually Laurel has to deposit a new can on his doorstep:

$$GAS \models \mu Z.(< Laurel\_deposits\_new > \mathtt{tt} \wedge [-Laurel\_deposits\_new]\mathtt{ff}) \vee [\mathcal{A}]Z$$

where $\mathcal{A} = \{Laurel\_deposits\_new, Laurel\_deposits\_old, Laurel\_collects\_new, Laurel\_collects\_old\}$,
$Hardy\_deposits\_new, Hardy\_deposits\_old, Hardy\_collects\_new, Hardy\_collects\_old\}$.

2.6 Formal Validation and Verification

In this section we discuss some approaches to the validation and verification of models, addressing some of the issues raised in the medical examples and then concluding with an examination of the 'Gas cans' as an illustration. Verification and validation activities usually involve tackling specific tasks with a selection of techniques. These tasks should be understood to operate within a wider framework of refinement such as that given in the next chapter.

Given an initial specification (or model) M_0 of the system, we need to *validate* it, i.e. check that it satisfies requirements, e.g., contains the essential sets of *traces* (sequences of events), has the appropriate responsiveness etc. Typically, a large class of specifications will satisfy the requirements, exhibiting the desired functionality and non-functional requirements. Where these requirements have been formalised, then depending upon the formalisms employed for the model and requirements respectively, we may proceed to attempt a proof. In earlier sections we have seen that non-functional properties, such as

safety, may be formulated in the modal and temporal logics and interpreted on the transitions system specified by process algebra, so we need systematic ways of showing whether or not such properties hold.

Bearing in mind the 'Safety Requirements Specification' in the Safety Lifecycle Model, we need to ensure that what was originally intended in the requirements is maintained subsequently. If a certain specification has been shown to satisfy the requirements, then further refinements, if they are consistent and complete, will preserve these requirements. Although there may be more granularity or detail and some change in shape, the structure is in some sense either equivalent or a containment. By following the development cycle given in Figure 2.3, we may attempt to refine a model stepwise towards implementation in some such manner, *verifying* each step.

The next subsections outline some notions of consistency and approaches to carrying out proofs for validation and verification.

2.6.1 Notions of consistency between models

In order that formal representations may be deemed consistent with each other there must be a relation that compares them. Such relations may be symmetric or asymmetric.

Regarding symmetric relations, an implementation will not usually be exactly the same as a specification, but if one takes a suitable view, then they may be considered *equivalent* with respect to sharing certain properties that have been abstracted out. *Equivalence* is a symmetric relation that can be defined in terms of elements belonging to a common class $\mathcal{C}$ (of objects) belonging to a set $\mathcal{S}$ of classes. The view chosen will determine whether or not processes are equivalent. For instance, a standard toaster and deluxe toaster may be deemed equivalent *modulo* (with respect to) a set $\mathcal{S}$ of breakfast utensil classes – {toasting implement, boiling implement, frying implement}. However, if we are more stringent and define a set $\mathcal{S}'$ of classes of toasters that reflect the level of functionality – temperature control etc. – then it is likely that the two toasters are not in the same class, i.e. are not equivalent *modulo* the new set $\mathcal{S}'$.

When the specification and implementation are in the same language, then one may define equational laws which induce equivalence classes. Such laws allow specifications to be gradually refined – through a sequence of elements of the same class that preserve perhaps the same observable functionality, but differ in other respects.

For process algebras such as CCS and LOTOS, notions of equivalence are defined in terms of behaviour. Different levels of equivalence are defined for processes in terms

of the actions in which they can engage in particular states. A useful equivalence is provided by defining a certain binary relation over pairs of behaviour expressions (or *agents*) – *bisimulation*, first described in [Par80]. The definition of these relations depends upon the notion of observation – some actions, representing those which we can witness or see from some viewpoint, are termed *observable*; others, which are internal to the system, over which we have no control, are *unobservable*. Accordingly, a label set may be partitioned into observable and unobservable actions.

We give below the definition for Basic LOTOS (see Appendix A for notation):

Definition A binary relation over behaviour expressions $\mathcal{R} \subseteq \mathcal{B} \times \mathcal{B}$ is a *(weak) bisimulation* if for any pair (B_1, B_2) in $\mathcal{R}$ and for any string s of observable actions,

1. Whenever $B_1 \stackrel{s}{\Rightarrow} B_1'$ then for some B_2': $B_2 \stackrel{s}{\Rightarrow} B_2'$ and $(B_1', B_2') \in \mathcal{R}$
2. Whenever $B_2 \stackrel{s}{\Rightarrow} B_2'$ then for some B_1': $B_1 \stackrel{s}{\Rightarrow} B_1'$ and $(B_1', B_2') \in \mathcal{R}$

Definition B_1 and B_2 are *observationally equivalent*, written $B_1 \approx B_2$, if $(B_1, B_2) \in \mathcal{R}$ for some bisimulation $\mathcal{R}$.

Informally, bisimulation states that, for any given pair of states in a relation, actions on one side can be mirrored on the other in such a way that after such actions, we arrive at a pair of states which also belong to the relation. This notion was subsequently adopted for CCS in [Mil89]. From it we may derive assorted equivalences such as *observation equivalence*, which we've just defined, *testing equivalence* [NH84] and *refusal equivalence* [Phi87], many of these notions being mutually alternative characterizations as shown in [Abr87, CH89].

Non-symmetric relations can be defined naturally from equivalence relations by just insisting that a relation works in one direction. Amongst such notions is one that takes just half of a bisimulation, called *simulation*. Non-symmetric relations are useful in reflecting that refinement is an activity that moves in a direction where, perhaps, details may be added or choices resolved that were left open by the specifier. Many of these relations for process algebra are *preorders* (i.e. reflexive and transitive), including trace inclusion [BHR84] and the testing preorder [NH84]. A few other relations, such as **conf** [Bri87], are not preorders but can also be valuable.

Having chosen a suitable relation, whether based on equivalence or some other, one must decide upon appropriate techniques for proving that the relation holds. Since it is usually the case that a whole class of specifications are equivalent, say, then methods employed are often designed to prove some kind of sufficiency, rather than exact equality.

2.6.2 Methods of Proof

Approaches to verification of desired properties or consistency may be classified according to the extent that they require interactive assistance from the specifier/designer as reflected in the two main approaches to proofs:

1. **Proof-theoretic** (or axiomatic): where specifications are written in or translated into the notation of a proof system in which theorems may be proved using, for example, equational reasoning and term rewriting.

   This approach is traditionally interactive, with activities that include studying the laws of a system itself; guiding a tool towards proofs, perhaps updating its proof capabilities along the way; and checking the validity of one's own hand written proof (*proof checking*). Included in this category are methods based on deduction where a verification problem in the given setting is not decidable, or at least no adequate decision procedure can be devised. In this instance, proofs have to be completed with intervention.

   A popular example of a theorem proving environment is higher order logic, a form of typed predicate calculus that can be based on λ-calculus. Such logic lends itself well to mechanisation: one system that has been devised is called Logic of Computable Functions (LCF) [GMW79], which in turn has subsequently been used as the basis of a popular tool called HOL[GM93]. Some applications of HOL are featured in a special issue of a journal [T.F95].

   In [MP81] it is proposed that transition system models of programs be coded as a theory Γ of temporal formulae. Provided that the axiomatisation is sound and complete, one can show whether or not a required temporal property holds as a consequence of Γ. Hence, the transition system representation gets sidelined in favour of the proof system.

   Theorem proving can also be used with languages that offer more structure than transition systems. A proof theoretic view of process algebra uses *equational reasoning* in which valid transformations are defined via equations as given in [Hen88]. Such definitions allow automation as *term rewriting*, and this is explored for the verification of LOTOS specifications in [Kir94].

   Properties of systems specified in process algebras have been proved using *compositional proof* methods working with a given notion of equivalence: a system is broken into components which are shown to possess certain properties that are together strong

enough to imply the desired properties of the overall system. For instance, a protocol has been verified using *relativized bisimulation* [LM86].

Another type of theorem proving uses a *(semantic) tableau*, a tree diagram constructed in order to check whether or not a set of statements are consistent by successively breaking down the given statements into simpler componenets, where the consistency depends upon obtaining contradictions or otherwise. Tableaux methods may be automated, in which case they underpin model-checking (see below).

2. **Model-checking** (state/process based): A system is 'captured' in some way by a machine representation, perhaps a finite automaton or possibly an infinite state machine. Since it is easier to manipulate systems with simple (or very little) structure, a process algebra definition is often transformed into an LTS representation that preserves a strong form of equivalence. Such a representation is generated by recursively applying the rules for transitional semantics. An algorithm, called a *model checker*, then can establish automatically and exhaustively whether or not desired properties hold for this representation (and hence for the original definition, if applicable).

   The initial work that was undertaken into model-checking is covered in [CES86] and [QS81]. Subsequent research and applications have been extensive: in [SW89b], the idea of a CCS process having a temporal property is discussed, supported by a correct model checker for the linear time mu-calculus of the general temporal logic. In [Bar95] model checking is performed on a specification of a microprocessor; and employed in [BA91] to gain assurance for a communications protocol of a real-time control system in the nuclear power industry. For verification, [FST92] presents a case study that performs stepwise refinement of process algebra, using bisimulation as the basis of the equivalence between successive designs.

   Apart from using model-checking to perform wholesale verification and validation directly on the specification, somewhat more implementation-oriented *testing* may be used. Testing is dependent upon how the system's behaviour may be observed in its external interaction. Tests may be derived from an initial specification, and the resulting interaction with the *implementation under test* (IUT) simulated. This offers quickly some initial indications of whether an implementation satisfies certain requirements. The types of tests possible depend upon the language used. This is covered in detail in chapter 5.

There is some overlap in the two approaches above and, indeed, either can be used for the checking of safety properties. Common to both is the need for efficient use

of resources such as hardware and time: algorithms should solve decision problems within reasonable bounds, using modest amounts of storage space. This issue is discussed (for model-checking) in [FM91, GRRJ89, GH93].

2.6.2.1 Simplifying the Computation

There are methods which are designed to enhance the proof activities above, often to simplify matters both conceptually and computationally. Two important principles that underline many such methods are abstraction and modularity. Thus, if a verification is being performed between two designs, then transformations may be performed on the respective representations to hide superfluous detail and decompose the system into more manageable chunks. This may be done such that the consistency between the original objects under comparison may still be checked. If validation is performed for some property expressed in one language with respect to a model in another, then abstraction may be performed on the model to yield the desired simplification.

There exist a variety of techniques to simplify the task of verification for systems. The techniques available depend upon the relation being checked. For instance, there are various methods of abstraction to choose from, of which those based on equivalence are well established, see e.g.s, [Zui89, Klu91] for ACP. Abstraction to show partial properties is conducted using the tool AUTO in [MV92a] where sets of actions are hidden to show some sequential properties of inputs and outputs to a communications protocol. In [AL89a] there is provided a modular specification method, where a proof rule is given that if components behave correctly in isolation then they will behave correctly in interaction with other components. Safety and liveness rules are discussed for such modules.

Those formalisms that explicitly model time are generally more complex, so abstraction is even more important in this case. Applications to timed formalisms include [DLG92] that uses a branching time logic RTL where the abstraction takes advantage of modularity. Also [Ost94] uses compositional reasoning to verify properties that are expressed in a real-time temporal logic (RTTL).

However, it is argued in [BBBC94] that time and abstraction can be conflicting, so a dual language framework is presented where behaviour and timing requirements are defined respectively as LOTOS processes and specifications in a temporal logic, QTL, that is based on RTTL. In particular it is proposed that to clarify the nature of abstraction from implementation details (termed *implementation abstraction*) there should be a separation of timing concerns into 3 categories that range from the abstract temporal ordering (*functional behaviour*) to explicit performance requirements for the implementation. This

implies that validation in the design cycle should be carried out in stages: early on, the more abstract temporal requirements for functional behaviour may be validated; later, explicit timing requirements can be checked for a specification in T-LOTOS. This separation may be incorporated in our model for formal refinement given in figure 2.3, where the sequence of successive requirements can be instantiated to correspond to this classification.

In the paper, such a specification is intended to be just a leaf off the main branch in the refinement trajectory, which is to remain within the scope of untimed LOTOS. Alternatively, one could treat the timed specification as some design item in the main refinement path, a view which appears natural and necessary if one wishes to avoid discontinuity in the development. A motive that supports this is the wish to validate the model, not some transformation of it; the timing requirements themselves may be introduced in stages in parallel with the model's development.

Whether or not such a specification forms a central part of the evolving design, there remains the issue of consistency between untimed and timed specifications. This seems resolvable since in [MFV93], it is shown that T-LOTOS is upwardly compatible with LOTOS by defining the time domain to be a single element, supported by a few appropriately defined equations.

One general approach that is gaining in importance is *symbolic verification*. This encompasses those methods that use symbolic representations of a system for the checking. Its computational advantage arises usually through being able to show proofs for a simpler representation, typically at some level of abstraction that is higher than that for an alternative representation that does some interpretation that reduces the structure and perhaps does some interpretation in the process. In particular, such an approach often proceeds at the level at which a design is expressed, be it in process algebra or some other symbolic language.

Symbolic methods may be further enhanced by the use of compact representations such as Binary Decision Diagrams (BDD's) which use Boolean expressions to provide considerable improvements in the number of states that may be stored. A state of the art for BDD's is given in [Bry95]. One application has been *symbolic model checking* in which a system is checked without interpreting any of the variables, using BDD representations of a version of the $\mu-$calculus [BCM$^+$92]. The efficient generation of such representations is presented respectively for CCS in [EFT91] and LOTOS in [Sis95], in particular they present a BDD encoding of transition systems. Note that in the case that the variables are defined on infinite domains, then the generated transition system would be infinite, so symbolic verification appears exceedingly useful here.

An important and typical context for abstraction is refinement: for instance, *action refinement* (replacing actions in a specification at a higher level of abstraction by processes of more concrete actions at a lower level of abstraction) is discussed for LOTOS in [CS93], where an extra refinement operator is introduced; laws that simplify CCS process terms whilst preserving temporal logic formulae are presented in [Bru93]. (A related task is *process refinement* (the refinement of the substructure of process definitions to provide more structure)).

More general property preserving maps between LTS are given in [LGS+95] which use Galois connections between LTS and function domains with pre/post conditions. Such abstractions are not based on equivalence. Another technique uses partial orders that abstract out from independently interleaved sequences and is shown to be computationally efficient [GW91].

Some recent work seeks to utilise symbolic methods in a project that combines theorem proving and model checking: the hardware description language ELLA[MC93] has been given formal semantics based on enriching automata and is in turn contained in a more general framework calculus of automata. Within this framework, symbolic verification of bisimulation is performed using a *state evolution rule* that defines a pair of logical expressions on abstract deterministic machines, whose structure enables the decomposition of the expressions into a set of first order verification conditions [BGMW95].

2.6.3 Tool Support

Verifications of even moderate size are very labour intensive, so automated assistance is highly desirable. A number of the references quoted above used tools to assist in proofs. The tools below are amongst some that are readily available, often public domain, and are well tried and tested. Some of these now employ nice graphical interfaces. No one tool is comprehensive: useful automated support really requires a collection of tools, possibly several collections. For instance, some tools perform a wide range of verifications, but require input of a certain format, necessitating perhaps the use of other tools to perform translations into this format. The list is far from complete and favours those with minimal intervention from the specifier, reflecting the view in this thesis that such an approach is more likely to be accepted in industry. For instance, a more interactive approach requires greater technical knowledge about formal languages which does not necessarily enhance understanding about the systems under scrutiny.

2.6.3.1 Theorem Provers

There are a number of tools to support this approach such as the Larch Prover for first order logic [GG89] and the Rewrite Rule Laboratory (RRL) [KZ95]. Equational reasoning needs more interaction as rewrite rules are developed and verifications carried out, but this can lead to a fine understanding of especially the kinds of assumptions which we make as designers. The Larch tool has been used in [CN92] for the verification of observation equivalence between two systems.

PAM (Process Algebra Manipulator) [Lin91] is a proof assistant that allows the equational laws to be encoded; tactics can be define to provide some automation, but this is limited. PAM is examined for Basic LOTOS definitions in [Kir94]. It has subsequently been refined to VPAM [Lin93] to cater for value-passing and uses symbolic methods that are similar to the work mentioned above on ELLA.

2.6.3.2 Model-checking and Others

The Concurrency Workbench (CWB) [CPS89] is able to check equivalences for CCS processes plus modal and temporal properties. However, earlier versions were limited in their potential to support other notations since they were unable to accept specifications that were not written in CCS or its variants. Subsequently, a common file exchange format has been defined, called FC2 [MdS93], which has enabled the integrated use of a number of tools.

The FC2 format enables the CWB to be used more easily in conjuction with the Lotos Integrated Tool Environment (LITE) [PvEE92] that was produced within the ESPRIT Lotosphere project. LITE has functionality that ranges from syntax and semantics checkers, to graphical G-LOTOS representations and a report generator, which together support well the development of specifications in LOTOS. LITE has limited model checking. Nevertheless, as Basic LOTOS is very close to CCS, there are tools which can transform LOTOS specifications into CCS, via FC2, e.g. more recent versions of M-AUTO that were detailed in [MV92b]. Verification of properties of Full LOTOS is generally limited due to the relative complexity of the language. The only facility available in LITE is a trivial reduction option in AUTO (see also [MV92b]).

LITE is stronger in its facilities for testing. One of its tools is LOLA [QPF89, Lla91] which provides for state exploration and test expansion facilities and has been used in the validation of communication protocols [CG93]. Another tool, SMILE [EW93], allows analysis of behaviour and ADTs through simulation performed using symbolic execution

event by event. However, such tools are generally insufficient for full formal verification.

However, another toolset, CADP, provides rather more facilities for verification. It consists of a pair of compilers CÆSAR and CÆSAR.ADT which translate a large subset of Full LOTOS to labelled transition systems [FGM+92] plus a model-checker, Aldébaran [Fer88] which can check for various preorder and equivalence relations using methods that use symbolic representations. There is also implemented an algorithm that generates models that are minimal with respect to strong and weak bisimulation. This tool also uses various enumerative verification methods, including some based upon the use of abstraction criteria that enable compositional reasoning. The CADP tools were employed in [Mou91] for the verification that an algorithm met the requirements for a reliable multicast protocol. This paper highlights how state explosion is revealed as a problem that can be beyond the computational abilities of a tool, thereby necessitating hand proofs. However, on further understanding of the system and the modelling language (LOTOS), sufficient proofs could be generated by hand with useful machine assistance.

Other tools for process algebras include: ARA [VS95] which has a way of visualising *global* properties of specifications (using LTS) – showing, for example, deadlock detection. This is dependent upon the computability of graph – techniques used for increasing efficiency include abstraction and decomposition, which do not need any special prompt from the specifier.

Some recent projects have moved towards providing tools that integrate various approaches: for example, PVS (Prototype Verification System) integrates theorem proving and model checking [ORR+96].

2.6.4 Validation Issues

In this section we briefly illustrate some concepts and tool application to the 'Gas cans' example.

Recall we wished to check that the specification does not deadlock. This is formulated as:

$$GAS \models \nu Z. < \mathcal{A} > \mathtt{tt} \wedge [\mathcal{A}]Z$$

Such a check may be carried out in a tool like CWB, but a quick simulation in SMILE reveals deadlock after just a few transitions: Laurel and Hardy move the cannisters without hindrance until they both decide to pick up from their respective ends (represented by a sequence `Laurel_collects`; `Hardy_collects` or vice versa). They meet each other

halfway down the garden path; according to our specification, they can only proceed by handing over their cannister, but the other person can't receive because they have their hands full – this is deadlock! Thus, some desired properties may often be shown *not* to hold by simple means, whilst showing that they *do* may require more effort.

Of course, in real-life this problem may be easily resolved *ad hoc* by a variety of ways, analogous to program 'debugging'. Unfortunately, specifiers of a safety critical system don't have recourse to this – they cannot afford to wait until the testing stage for design errors to be uncovered. Great care is required to ensure through formal analysis the necessary ongoing capability of events to be performed. (A solution to this is given in the Appendix).

This problem is essentially the same as one for CCS expressed in terms of communication buffers in Chapter 1 of [Mil89]. This is a tiny example, but it is just this kind of problem which lies at the heart of large and complex distributed systems.

We may use abstraction in the 'Gas cans' example by looking only for indications that everything is proceeding OK, viz that Laurel is collecting old cans, depositing new cans; Hardy collecting new cans and depositing old cans. i.e. we wish to abstract from the exchange process – do this in LOTOS using the *hide* operator.

```
hide deposit_old, collect_new in

Transfer[Laurel_collects_old, deposit_old, collect_new, Laurel_deposits_new]
 |[deposit_old,collect_new]|
Transfer[deposit_old, Hardy_deposits_old, Hardy_collects_new, collect_new]
```

2.7 Medical Examples

We provide a quick review of the use of formal methods for medical applications. These are rather sparse. The use of semi-formal methods such as functional programming are not listed, though it is worth mentioning a paper that reports the use of functional programming in (subjective) preference to Z for medical diagnostics, a project that was carried out under a strong safety-critical methodology that used techniques such as HAZOP, FMECA, and risk assessment [CBM+95].

Here are some of the projects:

1. Hewlett-Packard Laboratories carried out a couple of projects which involved formal notation – the formal notation HP-SL [Bea91] was used in the development of bed-

side instruments to monitor vital signs [LF91]; and the specification of input–output relationships over time for some safety-critical code on ROM [CBH91]

2. An appraisal of the use of Z to specify a family of instrumentation systems that was carried out around 1990 at Tektronix Corporation is given in [GD95]. It illustrates how formal methods may be scaled.

3. More recently, Z has been used in the specification of a control system for a clinical cyclotron [Jac93].

4. As computers play an ever increasing role in patient healthcare, the need for standards increases. Formal methods have been shown to enhance the quality of an international standard for medical device communications in [NC96, CNS96] (discussed below).

5. Another application of formal methods has been to analyse one of the most notorious software-related disasters in the medical sector – that of the series of Therac radiation therapy machines, of which one model (number 25) had software design errors that resulted in fatalities [Tho94, Kir95].

A noticeable feature of these projects is that the use of formal methods required the special support of management to go ahead; when this is not available, or lapses, then projects involving formal methods seem to fall away – as seems the case in the first two cases. However, work is ongoing in the use of formal methods for medical device communications, discussed next.

2.7.1 Medical Communications: background to Flexport

Embedded computer systems are increasingly common in intensive care environments where numerous medical devices are used to monitor patients' life signs and maintain essential physiological functions. Such distributed systems are dependent upon communication protocols. During the past few years, a significant development has been the emergence of a new international standard for Medical Device Communications, the IEEE 1073 Medical Information Bus (or MIB) ([Ins92, Ins94a, Ins94b] which is based on the existing ISO OSI 7 layer reference model for Open Systems Interconnection [Int84]. Introductory material to these standards is given in [GTHE89, Sha89, SW90].

The IEEE standard provides rules and guidelines for the connection of medical devices and host computers specialised for the intensive care environment. The major goal is the integration of all medical devices through "plug and play" where each component is

able to slot into a network without problems of compatibility and then communicate with other components according to the rules of the MIB. For hospitals, this will then enable with confidence the complete integration of all devices needed in their intensive care network, without being limited to proprietary systems. Through the flexibility of 'plug and play', medical staff will be able to choose from assorted manufacturers the best of each type of device, and so provide better care to patients.

Manufacturers will naturally be obliged to produce equipment which conforms to the standard. They will need to provide assurance that their products do indeed conform to the standard, especially the implicit safety requirements, hence the need for reliable software engineering techniques and the suitability of using formal methods. There is extensive literature on the application of formal methods to the analysis and design of communications protocols, see e.g.s [LM86, AL89b]. Further, its scope has been quite wide-ranging: [AHJJ93, ASvS89] and has even reached some reached some maturity [KvdLRS93]. However, like formal methods in other areas, its application in industry has been sparse, not generally integrated with safety analysis techniques, and there are especially few in the medical arena.

In this thesis, industry standard protocols are being used as the basis for investigating the application of formal methods to ensuring the overall safety of communications systems. Particular attention is being given to Flexport, a patented protocol (c) SpaceLabs Inc. for connecting a third party device [SpaceLabs89]. SpaceLabs is a leading developer of medical devices. The protocol will serve as a testbed for the methodology and techniques developed.

The research has been closely involved with the 'Medical Software Engineering Group' (MSEG), a multi-disciplinary research group with members from the Department of Biomedical Engineering at the Royal Brompton Hospital, The Department of Medical Computing and Informatics at the Royal Free Hospital School of Medicine as well as the School of CSES at Kingston University. The group consists of researchers in the areas of Computer Science, Biomedical Engineering and Electronics, some of whom have been on the balloting committee of the MIB. Associated work includes the generation of suitable prototypes for medical systems including the MIB. This has resulted in some papers that deal with verification: theorem proving is employed to examine the consistency of the MIB's data link layer of the MIB [CN92] through two views (intensional and extensional); and validation: it is shown how the confidence in such a standard may be enhanced through the specification of properties in temporal logic and their validation using model checking (in CWB) [NC96, CNS96].

2.8 Conclusions

In this chapter we have introduced the use of formal methods for safety critical systems, which started with a general systems safety view as regarded in engineering and successively focused on the safety aspects for software, leading to the Safety Lifecycle for software development. Within this traditional framework, we have developed explicitly the mathematics which appears to have the protential for analysing safety, expressing requirements and subsequently implementing systems. Due the vastness of the area, we have concentrated on concurrent systems and a particular twofold approach, using process algebra, viz LOTOS, and modal and temporal logics. It is a path which is chosen for its suitability for analysing the case study of the Flexport communications protocol to be treated later on in the thesis.

Chapter 3

A Framework for the Safety-oriented Formal Refinement of Systems

3.1 Introduction

In this chapter we are concerned with providing effective support for the refinement of formal models of safety-critical systems, which are intended to be developed towards eventual implementation. We investigate an approach that takes as its central perspective the use of formal methods since formal objects are to be the principal design items. A supporting framework is proposed that is based upon engineering principles, held together by close reference to the safety lifecycle model.

If formal methods are to be used primarily for the activity of generating safety requirements, then one may advocate (as in [dLSA95]) that formal methods be 'mixed in' with other approaches. However, here we are engaged in modelling and refinement activities, for which we suggest that greater emphasis be placed on the formal approach, which thereby becomes the 'main ingredient' that deserves special attention. The discussion is augmented by consideration of risk management to keep the focus on safety and of Configuration Management to maintain the smooth development of software items.

In order to motivate the discussion, we start by taking a closer look at the uptake of formal methods in industry.

3.2 Appraisal of Safety-critical systems and Formal Methods

It has been widely recognised that formal methods can play an important role in the production of safety critical systems, so what has the uptake been like?

Through the development of tools, formal methods have been applied on a larger scale. Indeed, their use for safety-critical systems has reached some maturity: articles such as [BS93a, CGR95, PA94] show how industry has not only recognised them as important, but also started investing in these methods, a situation re-inforced by the recent publication of a book [HB95]. However, the overall sentiment seems to be cautious and there are still misconceptions [BH95]. Closer inspection reveals that the application of formal methods to industrial examples has generally been sporadic [VvSP93] and in a rather static manner, sometimes viewed like 'plug-in' technology which, analogous to the comment about software 'fixes' (section 2.2.4), renders their use much less effective.

To discover more why this is the case, we choose to focus our attention on a specific area – communications, on which large systems in general are becoming increasingly dependent. Although this is not confined to safety-critical systems, its investigation reveals important clues.

There are substantial examples of using a formal description technique to specify complex communications systems, offering various insights into requirements for correct design. However, this has usually been conducted as some initial activity, rather separated from the main software construction. The *analysis* has often been rather modest – perhaps consisting of simulation and testing of the resulting large structure for certain selective behaviour, thereby proving just partial properties. Where greater rigour has been applied, the examples or case studies that have been chosen have usually been small and required *ad hoc* proofs by hand which, notwithstanding the issue of whether or not they may be scalable to larger systems, is a deterrent to all but specialists in formal methods.

It is arguable that the formal approach has yet to achieve its potential in any industrial area, for it has only been recently considered what are effective design *methods* employing formal notations to generate working products, as highlighted in [Bri92]. For concurrent systems, designers have commonly adopted process algebra or other formalisms such as SDL[Z.192] and ESTELLE[ISO89b] that are based on transition systems since they are supported, as we have seen, by some powerful tools. However, much less has been done or made publicly available on methods to manage projects in which formal methods play a central role – papers such as [Pen91] have offered only interesting glimpses.

One of the few projects that has made a substantial visible contribution (regarding the use of LOTOS) has been the Lotosphere consortium, which devised a tool-supported methodology to support the formal development of large systems [LOT92b, LOT92c], summarised as a book in [BvdLV95]. Even so, the methodology described therein assumes that only weak validation may be performed, through selective testing – which is inadequate to

show safety. Further, the central design activity of refinement remains little researched outside of broad considerations, though some literature is available and should increase as the results of one or two projects emerge [Bru94, For94, SBD95]. More general issues of making changes to formal specifications have also only been sparsely addressed [Kuh92, BW94]. Such incompleteness, especially as regards the stepwise development and validation, reduces considerably the impact of formal methods.

The use of formal methods for safety-critical systems presents a particularly 'hit or miss' picture. This is probably because, as highlighted earlier, it is difficult to ascertain what constitutes *justifiable assurance* of safety and dependability in relation to the software context. The safety lifecycle model helps, but there remains much hesitation (and so debate) regarding the role formal methods should play. One reason for this is that little has been done to provide a place for formal methods within systematic safety analysis as part of the software development. Where formal methods have been used for developing safety-critical systems, there has usually been little coherence, this being particularly the case in academic research, where the original motivation for a formal treatment of safety can be lost in theoretical niceties. As an illustration of the shortcomings, it sometimes appears in conference publications devoted to formal methods that safety-related properties have been discovered arbitrarily, even though there has been a real case study at hand. This often arises since the (formal) design cycle was not based around the provision of safety as required by users, so there was no inherent view of safety. Consequently, formal safety-related requirements have often been considered *ad hoc*, and *post hoc* (in retrospect).

3.3 Strategies for a coherent approach

A general overview for safety-based development of systems with embedded software has been presented in [Lev86, Lev91], which integrates many of the industrial safety techniques into the framework for producing safety-related software. In this and other similar papers, formal methods are part of the consideration, with valid talk of the need for their selective use, but a systematic framework that defines the role of formal methods is not made explicit. Even though some of the support for safety analysis has been expressed formally, there is little available on how, for instance, notions of hazards and risks affect formal models and designs themselves.

The provision of assurance needs to address at the outset systems safety analysis and safety-related requirements, and then follow this through the entire development. Hence, if formal methods are used, it is preferable to capture the analysis and requirements

formally. The formal development should also be properly documented to constitute part of the safety case, which is integral to the assurance for a coherent whole. Only when it can be shown that safety arguments can echo through the formal model, and subsequently in code, can they be properly justified. The ability to follow any arguments contributing to the justification is called *traceability*, which requires a wide framework in order to support the production of safety cases. Such support for software in general has been provided in [FJMP94, McD94], but this is only semi-formal, and does not specifically cater for formal models. Indeed, apart from the work mentioned below, there appears to be little in the way of guidelines for recording the specifications, their relationships and any other formally related material: the tacit assumption is that they can be treated like any other design object.

In response, there has been some work that provides a formal analysis of safety requirements. In [SdLA95, dLSA95], there is the application of formal methods to the analysis of the requirements for systems in general, and process control system, in particular. The work, which has steadily refined earlier research [dLSA91], is firmly rooted in system safety practices: it provides a common formal basis for the safety analyses and requirements through an Event/Action model for reactive systems, based on Petri nets [Pet81]. These papers also provide traceability of specifications in the form of *safety specification graphs* (SSG). These graphs record links that range from the accidents through to the specifications generated by the requirements analysis, and also allow the formal definition of logical relationships between these objects. A structure is provided that is generated according to a means of abstraction, namely decomposition: in the SSG, accidents lead on successively to hazards, safety constraints (which negate the hazards), safety strategies that maintain the constraints, interface safety strategies and finally control system strategies.

Qualitative risk analysis may then be employed to analyse the safety specifications: *preliminary analysis* checks for consistency between different levels of the SSG and validation of hazards against accidents, both provided using Event/Action model; *vulnerability analysis* of the specifications is used to identify the circumstances under which the specification is unable to maintain safe behaviour, for which failure analysis is conducted using FTA. This is mentioned as being used "in conjunction with formal analysis", though little other information is given; no procedure is made explicit for linking the two. Finally, it is reported that this methodology has been applied to a case study, and found to be particularly effective for systematic analysis [dLSA94].

Another common formal semantic basis for traditional safety analysis techniques that is well established is the Common Safety Description Model (CSDM) [BCG91, Gó94].

This work has gone deeper into scrutinising the safety analysis techniques themselves, providing formalisations of FTA and Event Tree analyses. In the process, this work has frequently highlighted how the safety analysis techniques themselves can be ambiguous. CSDM is discussed in more detail in the next chapter.

Other work on safety analysis has been reported in [CFH95], which has the distinction of integrating formal and informal methods into one methodology. This strategy is growing in favour within software engineering and is called *Methods Integration*, whose philosophy is that each technique has its relative merits and that they may be properly realised through being anchored in appropriate phases of the development.

The paper has a 3 language approach: Ward and Mellor Essential Models [WM85] are used to model in turn the system environment and then the system behaviour; after the environmental model is deemed to be sufficiently mature (having satisfied requirements generated after environmental HAZOP), the behavioural model is constructed and then translated into SCCS, a version of CCS that supports true concurrency [Mil89]. The environmental model is used to generate informal requirements for the behavioural model, which are subsequently translated into a branching time temporal logic, whose properties can be checked on the SCCS model using the Concurrency Workbench. Note that the paper refers to this activity as *verification*, whereas we term it *validation*. A particular feature of the validation of formalised requirements is the use of local model checking, which uses *proof tree analysis* that decomposes a formula expressed in logic into simpler component formulae such that the truth of the original formula holds if and only if every leaf formula is true [SW89a]. In addition, a process model provides a useful framework in which to conduct these activities, specialised to the context.

The paper is mainly concerned with building models for system analysis rather than the refinement of models towards implementation. A drawback of the approach is the extra work involved in using Ward and Mellor as a stepping stone to SCCS representations. It would also be useful if the process model could be related to a general safety lifecycle model.

3.3.1 Summary and scope for this thesis

In summary, work on safety requirements analysis has been quite thorough and this has increased the assurance of the dependability of software components. Design cycles have also been proposed to support these activities. However, less consideration has been given to how to relate these requirements to system models as they undergo refinement

and the provision of support to ensure that these refinements continue to have the required properties. This latter consideration also needs a larger lifecycle perspective: as yet, there is no discussion that makes explicit the needs for formal development in the context of a standard safety lifecycle model.

Hence, this thesis has two main thrusts: first, it attempts to build on these kinds of approaches, by devoting effort to a discussion of the general issues that occur for formal development in the context of a safety lifecycle model (figure 2.2). We analyse (in this chapter) the lifecycle model and propose modifications or elaborations, where deemed appropriate for the formal needs: this is done systematically by examining the respective stages of this lifecycle (which we denote by enclosing in boxes) for their amenability to formalisation. We also consider the suitability or otherwise of some common means of traceability used in the engineering and software settings. At a general engineering level, we consider the practicalities of a risk management log as a means of recording activities of formal development; for the software developer, we consider software configuration management (henceforth abbreviated by *CM*), invoking a more theoretical look, reflecting the wish to exploit the advantage of preciseness that may be defined in the relationships between formal objects.

Second, we focus (in subsequent chapters) on the techniques to effect the safety-oriented refinement of formal specifications. We choose to deal specifically with issues of concurrent systems, devising procedures and enhancing techniques which we shall later test in relation to Flexport. In particular, an evolutionary procedure enables requirements to be developed in stages and related to a model as it develops, suitable for any formal approach in the context of safety being central to user requirements. Such an approach can also facilitate a smooth transition into implementation, so that safety requirements do not become subsequently lost.

All this allows for more flowing and effective use of various well-established formal techniques – such as Conformance Testing, which is extended in Chapter 5. We also show how and where aspects of the research mentioned in the literature can be integrated within this overall scheme.

3.4 Foundations for safety-based development

We describe here the system setting, including basic concepts for the formal development of safety-related software. Although the treatment is geared towards *safety* analysis and subsequent generation of safety-related requirements, various other kinds of

requirements (economic, environmental, ... etc) can be treated similarly.

3.4.1 Introductory concepts

The first requirements come from the customers/users who are looking for a system that will meet their demands – which we denote by (USER-REQS). In response the system developer will generate a set SYS-REQS of requirements for the perceived system, S, which should be produced in some document (the requirements definition – see below).

The nature of user requirements and system requirements may be illustrated through the example of a Patient Data Management System (PDMS) for a hospital. In this instance the users are hospital staff, whose requirements may include an efficient system that can handle large volumes of data; a flexible setup that allows access from different parts of the hospital, especially the Intensive Care Units, Operating Theatre, Administrative offices, etc; the system should be easy to use; the whole project should meet a certain budget; and so forth. These requirements are drawn up in terms of the working practices in the hospital.

The hospital managers may invite tenders from various companies that deliver such systems. In response such a company may take the list of user requirements, try to clarify what is meant and then develop a set of requirements in terms of the technology that should meet their needs. These are called system requirements and they are produced by the system engineers in consultation with the users - the developer needs to understand what the users want and the users should understand how the system developers will meet their needs.

To meet the requirements in this case, any such system must include a network, component hardware and software and other system parts. For each of these parts requirements may be drawn up: for example hardware requirements may insist that data must be be able to flow between multiple platforms - the PC in the Patients' records office through to the console in an Intensive Care Unit. Part of the systems requirements would be the communications system; for these the system developers may inform the hospital managers that a properly integrated network may be achieved by implementing some recognised standard. Hence, an agreement may subsequently be reached requiring perhaps conformance to the Medical Information Bus. Thus these may be taken as system requirements.

In our context we consider the production of S as a process of refinement from an abstract formal model to an eventual implementation. S is typically part of a greater system Σ with wider boundaries. Safety analysis is expected to have been conducted for Σ, with its scope extending throughout its components or subsystems. Where we refer without

qualification to a 'system', it is S we have in mind. We define for our context the following general terms:

- *requirements definition* – a document detailing what is required by the users (USER-REQS) and a proposed system which sets out system requirements (SYS-REQS) in terms of services, constraints and goals.
- *design* – an ongoing process which starts with a conception of a system to meet the requirements definition and leads to a detailed definition of how the system can be implemented
- *verification* – the task of checking consistency within or between objects in the design
- *validation* – the task of checking that some (formal) representation of user requirements actually meets these requirements

Safety requirements are those requirements for S which are derived from that part of the safety analysis which has a bearing on S. The determining of safety requirements is an activity which follows on from the safety analysis. These requirements for a model may be partitioned into two: *functional requirements*, which are specified as part of the requirements of normal operation, and *non-functional requirements*, which have been derived specifically from safety analysis. In particular, safety requirements should require that a model:

- address the occurrence of faults discovered in hazard analysis
- provide, where appropriate, *methods of control*, being design strategies for reducing the risk from hazards.

Fault trees model hazards that arise and propagate both through the occurrence of undesired events and through normal system operation. The way that a hazard propagates up a tree may well be simply a result of reliable behaviour of some system component that is included in the functional requirements specification. If the hazard is to be removed then, as this example illustrates, one has to determine the relationships between the hazard causes in order to see where and how it can be dealt with.

This approach can be seen consistent with the 'Safety Requirements Specification' phase of the standard Safety Lifecycle Model by regarding the 'non-functional' requirements as *methods of control* to achieve the integrity required of the 'functional requirements specification'.

As an illustration of the distinction between functional and non-functional requirements, and the kind of decisions that need to be made regarding methods of control, consider the refinement sequence of models $M_i (i = 1, 2, ..., n)$ for a communications system. It may include the following requirements:

1. M_1 is to satisfy the functional requirement that data x is transmitted from station X to be received as data x at station Y.

   Hazard analysis, represented in the fault tree, reveals that there may be an error in the data received at station Y, having been propagated from station X. Hence ...

2. M_2 is to satisfy the non-functional requirements:

   (a) errors are possible during transmission of data (fault inclusion);

   (b) if there is an error in data x at station X, then a method of control m_c is introduced before it gets transmitted further. (fault tolerance)

In this example, a fault tree can determine several possible sources of error. However, here there is no stipulation about errors appearing at station Y, indicating that a choice has been made to control the hazard at a lower branch of the tree.

3.4.2 Safety-related principles

In order to justify reliance on the system we are developing, we are guided by principles for the provision of safety-related control systems as provided in draft standards by the International Electrotechnical Commission [Int91, Int92]. These principles, listed below, are explained in an article on Programmable Electronic Medical Systems (PEMS) in [Bib95], which is a useful reference basis for work on Flexport. We relate the principles to our context:

1. *safety is considered in the context of the whole system* – the analysis is 'top-down' (or deductive) from the identification of hazards as experienced by the medical staff and patients, and reflected in Fault Tree Analysis (FTA) [VGRH81]; and 'bottom-up' (or inductive) through Failure Modes, Effects and Criticality Analysis (FMECA) [Bri91]. Another key method is the use of HAZOP [Che87], now tailored for Programmable Electronic Systems [Def95], for which an overview is presented in [CBM+95].

2. *a development lifecycle is used* – the Safety Lifecycle Model

3. *a risk management process is used* – closely tied to where risks apply with respect to pre-defined baselines

4. *risks are required to be As Low As Reasonably Practicable (ALARP)* – to be minimised through various measures, but particularly formal methods

5. *safety integrity is used as a measure of the likelihood of a system performing safely* – computing safety integrity is difficult. However, at least for a formal specification, we expect formal proof to contribute to high levels of integrity.

3.5 Analysis of the Safety Lifecycle Model with regard to Formal Methods

The initial development of the system software is in a formal setting – it is this formal system that is to be refined towards implementation; it is likely that the software system will later become increasingly less formal, but nevertheless in its initial stages, we treat it as completely formal. In keeping a focus on safety, we must consider hazards and risks throughout the development process. In this section we treat in turn each stage of the safety lifecycle model and discuss the scope for formalisation. We also provide some medical examples, which help to provide suitable background for Flexport.

Part of the process involves establishing whether or not there exists a formal language to capture these concepts. If so, we can formulate the informal requirements. We have cited in Chapter 2 some common formal definitions of *safety-related properties*, but they may or may not correspond to those in the engineering setting. Further, those properties that can be formally defined may be insufficient to cover the kinds of properties associated with hazards, say. We consider this issue of completeness below, after respective introductions.

3.5.1 Overview of Hazards and Risks

Hazard Analysis, Risk Assessment

The System and its Environment are analysed to locate hazards and determine the nature and severity of any potential risk. The levels of risk are used in the stipulation of measures that need to be taken for the delivery of a system on which we may depend for safety (see sections 2.2.1 and 2.2.2 for discussion).

A key component of the development is the management of risks arising from hazards – it must be made clear what risks there are and how they are handled. We intend the risk management to proceed with the following steps, which in practice should be iterated throughout development:

1. Identify and list hazards
2. Use FTA to recursively identify hazard causes (other hazards) until root causes are established; use FMECA to identify further hazards
3. Estimate risks, making use of risk charts
4. Introduce appropriate Methods of Control
5. Plan CM baselines
6. Compile Risk Management Log

3.5.1.1 Hazard Identification

For generating a list of hazards which are typical in communication, it is most useful to have a broad view which takes in the experiences of patients, medical staff and engineers as well as more theoretical views. Asking nurses about the problems they have encountered may point to hazards not anticipated by a software developer, whilst some may point to other issues such as training and HCI. Assurance that the protocol has been designed to ensure a safe system is partly provided by addressing and responding to such experience.

Determining a complete list of hazards is problematic because some hazards have unknown cause and unknown effects, especially where people's behaviour is involved. There are methods which facilitate the process for programmable electronic systems derived from HAZOP. One such technique is SHARD [FJMP94], which incorporates the extensive research that has been undertaken into developing classes of *guide words* which suggest possible system failures. SHARD uses the following failure classes:

Service provision: *Omission*, *Commission*

Service timing: *Early*, *Late*

Service value: *Coarse Incorrect*, *Subtle Incorrect*

In the case of safety-critical communication protocols, we require reliable, constantly available, and accurate data *transmission* (relating to signals and transmission me-

dia). Dependent upon the communications channel, is data *communication*, for which we require the reliable, constantly available, and accurate communication of information. We may combine the three failure classes into two – *provision* and *timing* – and enrich them to give a classification given in the appendix.

3.5.2 Formalizing hazards for requirements analysis

The purpose of this section is to give a (formal) overview of the general requirements of formal languages for the analysis of safety and the generation of requirements for software models. We start by considering the requirements capture and draw up a formal model for establishing the relationship between hazards that have been identified and the universe of formal logic-based languages. It is immediately evident that any hazard that is identified may be given a formal denotation, but its meaning has to come from some finer structure; further, the ability to do reasoning depends on a language that is able to express relationships and prove properties reflecting real world situations between the hazards.

Thus, the problem of existence appears to have two parts: the need to express both individual hazards and the kinds of relationships between hazards. As a system is typically a multi-layered structure, then a hazard is often defined in terms of its components, which may well include hazards themselves. So the problem of existence is defined by the ability to express relationships: *horizontal* – between hazards at the same level of abstraction; and *vertical* – in terms of lower level components that make up a hazard.

Given common notions of hazards, there need to be considered some general criteria for suitable formal languages.

1. A well-defined syntax

2. language has a proof system that is sound and complete

Ensuring safety may be variously characterised in response to the hazards. A general approach is to consider each hazard in turn, analyse it, and then determine a set of requirements and subsequently show that a system satisfies these requirements.

3.5.2.1 The Hazard Existence Problem

We present the formalisation of two types of the hazard existence problem: first, in terms of whether hazards at a given level of abstraction can be modelled implicitly in terms of internal structure; second, in terms of sets of hazards for which a language can model all the required relationships between denoted hazards.

Note that these definitions span the 'informal' and 'formal' worlds, so they are semi-formal definitions, dependent upon subjective interpretations on words such as 'suitable' (always a potential source of divergence for validation). First we give the 'vertical problem'; and then the 'horizontal problem', where we take the view that some given hazards are atomic units, thereby obscuring their internal detail.

Let $\mathcal{A}$ denote an alphabet of symbols and let $\mathcal{H}$ denote the set of documented hazards h, where $h \in \mathcal{A}^*$ is just a string of symbols. Each h may be characterised through an appropriate formalisation of the properties; its significance arises in relating it to some context – a formal model. This raises the issue of what to choose in the way of languages for properties and the system model respectively. Where these are underpinned by a common formal semantics, the relationship can be made directly: in systems such as the modal mu-calculus, introduced in Chapter 2, the underlying framework is that of labelled transition systems in which properties are actually *identified* with sets of states of the system, though, in practice, a system model may have to be explored before it is revealed whether or not some hazardous states exist.

Here we concentrate on the formalisation of properties. Let Γ denote a formal language and let $\mathcal{H}$ denote a set of hazards. We define a simple valuation $v_\Gamma : \mathcal{H} \to \{\mathtt{tt}, \mathtt{ff}\}$ by: for each $h \in \mathcal{H}$, let $v_\Gamma(h) = \mathtt{tt}$ if the property h may be logically formalised in Γ, $\mathtt{ff}$ otherwise. Let $\mathcal{H}^+ = \{h \in \mathcal{H} : v_\Gamma(h) = \mathtt{tt}\}$. Since it is the case, that for any given hazard, it may be viewed as either atomic or made up of components, this represents, in effect, the ability to model vertical relationships.

In order to make more use of structure, let $\mathcal{H}$ be partitioned into n distinct viewpoints $\mathcal{H}_i$ (for instance, levels of abstraction), i.e. $\mathcal{H} = \bigcup_{i=1}^{n} \mathcal{H}_i$. Suppose that we have also a set of informal relations between the hazards, which we denote by $\mathcal{R}$. As $\mathcal{R}$ consists of horizontal and vertical relations, we may partition this set as follows: $\mathcal{R} = \mathcal{R}_V \cup \mathcal{R}_H$, where $\mathcal{R}_H = \bigcup_{j=1}^{n} \mathcal{R}_{H_j}$. We define that Γ is *complete with respect to* $(\mathcal{R}, \mathcal{H})$ if every relation in $\mathcal{R}$ may be formalised in Γ. In the case that Γ is not $(\mathcal{R}, \mathcal{H})$-complete, we define the *closure* of $(\mathcal{R}, \mathcal{H})$ in Γ to be the set of $\mathcal{H}$-maximal subsets $\mathcal{K}$ such that Γ is $(\mathcal{R}, \mathcal{K})$-complete.

Using the partitioning, we can be more precise about the level of completeness. Suppose that $\mathcal{H}$ has been partitioned as above, with $\mathcal{R} = \mathcal{R}_V \cup \mathcal{R}_H$, then Γ is *(horizontally) complete with respect to* $(\mathcal{R}, \mathcal{H}_i)$ if every relation in $\mathcal{R}_{H_i}$ may be formalised in Γ. In the case that Γ is not horizontally complete with respect to $(\mathcal{R}, \mathcal{H}_i)$, we define the *horizontal closure* of $(\mathcal{R}, \mathcal{H}_i)$ in Γ to be the set of $\mathcal{H}_i$-maximal subsets $\mathcal{K}$ such that Γ is $(\mathcal{R}, \mathcal{K})$-complete. Similarly, for the vertical relations: Γ is *vertically complete with respect to* $(\mathcal{R}, \mathcal{H})$ if every

relation in $\mathcal{R}_V$ may be formalised in Γ. In the case that Γ is not vertically complete with respect to $(\mathcal{R}, \mathcal{H})$, we define the *vertical closure* of $(\mathcal{R}, \mathcal{H})$ in Γ to be the set of $\mathcal{H}$-maximal subsets $\mathcal{K}$ such that Γ is $(\mathcal{R}, \mathcal{K})$-complete.

Hence, Γ is horizontally complete with respect to $(\mathcal{R}, \mathcal{H})$ if and only if $\forall i \in \{1, 2, ..., n\}$ we have that Γ is horizontally complete with respect to $(\mathcal{R}, \mathcal{H}_i)$. Further, Γ is *complete with respect to* $(\mathcal{R}, \mathcal{H})$ if, in addition, Γ is *vertically complete with respect to* $(\mathcal{R}, \mathcal{H})$.

For this setting we may then define a notion of safety in terms of two factors: the ability to model the informal hazards and their relationships and the demonstration that the formalised property (the negation of the hazards) holds for the formalised system:

Definition (*System safety*)

Given a set of relations $\mathcal{R}$ has been defined over a set of hazards $\mathcal{H}$, then a system S that is formally developed in Γ is defined to have *system-total safety* if

1. Γ is *complete with respect to* $(\mathcal{R}, \mathcal{H})$
2. $\forall h \in \mathcal{H}$, $S \models \neg h$, where $\models$ denotes a satisfaction relation between representations of a system and properties.

Various weaker relations of satisfaction may be defined depending upon the completeness of the coverage of hazards and the extent to which these hazards are shown to not hold for the system representation.

3.5.2.2 Reasoning about Hazards

We continue the discussion by making explicit the kinds of relations that may exist between hazards. In the safety lifecycle model, given a hazard, various techniques are used to identify their contributing causes, stimulated by guidewords. They can arise through temporal chains of events or, perhaps, their simultaneous occurence. They may also be due to the nature of their internal structure – which may be termed *subhazards*. Throughout the process the analyst has to reason about how all these hazards are related, for which FTA is one of the most useful means.

In general, a formal approach needs to:

- establish some formalisation of temporality, or perhaps more concrete notions of time, where hazards are temporally related;

- where a hazards is composed of subhazards, then use decomposition on the formalised hazard. Ideally, any properties ϕ should be expressed in some 'normal' form – as a conjunction of properties $\phi = \wedge_{i \in I} \phi_i$, whence ϕ is true if and only every component ϕ_i is true.

FTA is perhaps the most well established technique for formalising these relationships (see Chapter 4): it can be used to analyse the formulae themselves to establish how the overall formula may be negated, perhaps aided by compositional reasoning as given in proof tree analysis. Underpinning such activities is the consideration of to what extent the *discovery* of causes should be carried out in the informal or formal worlds, which depends upon the level of detail in the system models. For effects, HAZOP deviations can be determined using FMECA which can reveal previously unconsidered hazards.

In practice some hazards cannot be negated, though may be safely tolerated, and hence the above definition of safety is rather simplistic and somewhat inaccurate. A more sophisticated view (which is treated in the next chapter) is to consider more general requirements generated from hazards, so for any given hazard h, a set of required properties $\phi(h)$ may be defined, where once again one has to consider the issue of completeness since the requirements are driven from analysis of the real world situation. Further, proving all the properties does not imply that the safety of the overall system is completely ensured, since it is generally impossible to prove such completeness – how do we know that we have anticipated every hazard?

The structuring methods we have described are illustrated for hazards, but may be applied more generally, including to the SSG's of de Lemos *et al*, which record several different kinds of activities. The SSG's are produced in an incremental manner as the safety analysis/requirements phases develop. Applying the partitioning scheme above can facilitate especially the reasoning about completeness of the SSG's. We apply it to the formalisation of fault trees in the next chapter.

There are the following additional considerations.

1. New hazards can be discovered in the real world, which should then be considered for incorporation in the formal logic language

2. New relations may be discovered in the real world; the formal language should then be checked for its ability to model these; and the closure to be reassessed accordingly. (Although there is the notion of 'kicking away the ladder' once the formal scenario is settled upon, in reality, new information may always be forthcoming.)

3. All such changes need to be recorded by some means

In summary, even using a simple view, the discussion of completeness through the use of partitioning indicates that formal languages require tremendous scope in their abilities to model the desired properties and to effect proofs. It provides a useful motivation for investigating the relative expressiveness of different languages.

Progress in this regard is generally in its early stages, though some significant effort has been made to introduce more structure into state-based descriptions in the definition of RSML (Requirements State Machine Language), a functional language that allows compositional reasoning. Indeed, using this language, the issues of consistency and completeness are effectively treated in [HL96]. The tradeoff is that the semantics restrict the expressivity to some extent, for instance non-determinism is not allowed, yet even so a U.S. airborne collision avoidance system is substantially validated.

3.5.2.3 Example: Insulin Delivery System

We illustrate some of the ideas using an example of an insulin delivery system which is described in Chapter 21 of [Som92], where there is some discussion of hazard analysis. Although such a system is not from the ICU, the kinds of hazards in this example are likely in many intensive care systems such as intravenous pumps. First, we give a short introduction.

Those suffering from diabetes require supplementary doses of insulin on a regular basis to ensure that they have the right levels of glucose sugar in their bodies: low levels can quickly have serious effects on the brain, whilst ongoing high levels may give rise to problems in the eyes and kidneys. Doses also have to be carefully controlled to allow for the fact that the absorption and effect of the insulin is a function of time.

Advances in technology through the use of microsensors allow the level of blood glucose in the body to be constantly monitored. Hence, this information may be transmitted to a pump and dosages given accordingly using a needle attached permanently to the skin.

We now illustrate some of the formalisations of hazards and their relationships using this example. Note that the questionnaire in Appendix H should help this analysis by eliciting through experience further hazards and also hazard causes. In [Som92], Somerville gives the following list of hazards, to which we add denotations **h1** to **h7**.

h1: Insulin overdose

h2: Insulin underdose

h3: Power failure due to exhausted battery

h4: Parts of machine break off in patient's body

h5: Infection caused by introduction of machine

h6: Machine interferes electrically with other medical equipment such as heart pacemaker.

h7: Allergic reaction to the materials or insulin used in the machine.

In our formalisation, we consider first how to model the structure of the hazards themselves. In this context, the modelling of the first two hazards is impractical in a process algebra, but may instead be served by some state-based language such as VDM or Z. First some variable may be defined over the real numbers to measure dosage, but then we must ask, "What is an overdose (underdose)?". Such a consideration may lead to defining upper and lower bounds on quantity and the definition of dosage in terms of a function on a time domain.

For treating the hazards and their relationships, we look at spacial and temporal relationships. For example, we can analyse each statement and decompose using event splitting, triggered by conjunctions such as "due to", "resulting from", "caused by".

For example, for **h3**, "Power failure due to .." is a signal to consider all the causes of power failure and may be expressed as a disjuntion of causes:

$$h_3 = h_{3_1} V h_{3_2} V ... V h_{3_n}$$

where h_{3_1} = "battery is exhausted"; h_{3_2} ="fuse blown", etc.

The systematic means for determining causes - spatial and temporal - is fault tree analysis. In this case, we may abstract out from the details of what states actually constitute the top level hazard and develop various logical relationships that may be best served by some temporal logic. Hence **h1** and **h2** may be combined into one top level fault, yielding a fault tree, part of which may be as in Figure 3.1 (reproduced from [Som92]).

For the analysis of the tree, we select one of the causes of the top-level hazard, viz 'Incorrect Sugar level measured'. This has in turn as a cause 'Sensor failure' - a sensor might have jammed at a certain value resulting in no change to the sugar level measured during some fixed period. In this case, it is more appropriate to express this hazard as a temporal relationship rather than a simple disjunction. Further, if one wishes to be strict about duration in terms, then the logic ought to have an explicit means for measuring time.

These interpretation issues in Fault Tree Analysis are discussed in more depth in the next chapter.

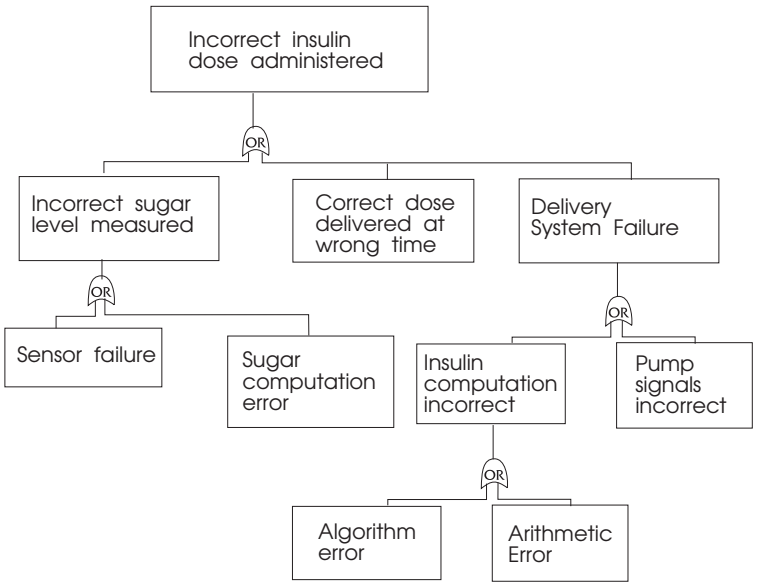

Figure 3.1: Fault Tree for Insulin Delivery System

3.5.3 Safety integrity

Safety integrity is defined as "the likelihood of a safety-related system achieving the required safety functions under all the stated conditions within a stated period of time" [Wic92]. For the formal scenario, one needs to determine what is meant by 'under all the stated conditions' and 'within a stated period of time.' This will depend to a large degree on the interpreation of the safety analysis.

The provision of integity levels is defined by:

Safety Requirements Specification

This specifies the requirements that vary in strength according to the integrity required of respective functions. It has two parts: *Functional Requirements Specification* and *Safety Integrity Requirements Specification.*

Functional Requirements Specification

" ... identifies which functions of a system are safety-related and in what ways."

This process requires that risks associated with hazards are computed for their bearing on the various functions of the system. For each system function a map can be defined that relates it to a set of risks.

Risks are estimated and may be quantified as follows: let $\mathbf{N}_r = \{1, 2, ..., m\}$, where m is the number of risk categories. Let Θ be a set of risks. Now define a valuation

$v_\Theta : \Theta \rightarrow \mathbf{N}_r$ by: for each $r \in \Theta$, let $v_r = n$, where r has been (informally) assigned risk category n. Let F_S denote the set of functions for a system S. Now define a relation $\wp \subseteq F_S \times \Theta$ by: $\forall \theta \in \Theta$, $\forall F \in F_S$, F is assessed to be related to some risk θ if and only if $(F, \theta) \in \wp$.

Treating the *type* of relationships is a more complex process, which may be determined from the hazards analyses since risks provide only assessments or quantifications of the potential consequences of hazards.

For some systems, such as a medical communications protocol, virtually all aspects of the system may be safety-related, contributing to the transmission of vital patient data, which must be accurate. According to the risk assessments, we may pick out certain salient functions to which we pay particular attention, which is given by the next phase of the lifecycle.

Safety Integrity Requirements Specification

" ... begins with the functions identified in the Functional Requirements Specification; it specifies for each of those, the required safety-integrity levels."

In the case of communications protocols, system integrity is in terms of correct and timely communication of data.

The purpose of these integrity levels is to ensure that appropriate risk reduction is effected, so that at a minimum a 'tolerable' risk is achieved. For the purposes of the formal treatment, we need to ask what this means, i.e. what constitutes a tolerable level of risk in a model? This amounts to determining what set of requirements must be satisfied so that the risk is adequately covered. As already discussed for hazards, formalising requirements is a problem of specification, dependent upon the expressivity of the formal notation(s) used for expressing properties, whilst demonstrating that requirements hold is the problem of relating models to properties.

Indeed, such requirements prompt further investigation into formalisation since they will be reflected in general terms by varying degrees of rigour of method – both informal and formal methods – to be used in the methods of control (see below). This kind of classification can be applied to the formal setting itself: the type of validation is directly determined by the interpretation of notions such as liveness and fairness and indirectly through notions of refinement such as *conformance*, a binary relation between specifications which stipulates that some behaviour exhibited by one must hold in the other (see Chapter /refch:conftest). The formal definitions must

be validated for their correspondence to the degrees of satisfaction as stipulated by the safety integrity requirements. This should also involve a thorough analysis of the proof techniques to be used and what is expected from them.

Transferring grey, sometimes fuzzy, notions (of degrees) into traditional propositional logic is a difficult matter since outcomes to decision problems are generally black, white or unclear, (or 'Yes'/'No'/'Undecided'). In view of this, it becomes evident that the issue of completeness is important, for it offers scope for shades of grey. However, since we are typically dealing with very high levels of integrity, such philosophical considerations are obviated by simply stipulating that total completeness is expected. This highlights the fact that the flexibility really lies in the *choice* of requirements which are to be proved.

Looking at this from the side of the various formal techniques, we have that different kinds of decision problems can be classified according to strength. Hence, for a minor requirement which may confine its attention to just a small part of the system, it may be sufficient to perform simple finite reachability analysis to test for a property. On the other hand, a major requirement may require a property to hold in *all* system states (the formal notion of safety defined in the previous chapter), which would require a powerful state exploration method.

The amount of effort we put into proving properties should be proportional to their importance as regards ensuring safety. In the lifecycle model, safety integrity measures this importance and there are defined 5 levels to reflect this, ranging from 1 (very high integrity) to 5 (very low integrity). In the formal setting, there is an analogue to this: a suite of partial tests to check for specific behaviour may be considered as providing weaker integrity than the verification of an implementation relation, say. Hence, notional relationships may be established between levels of integrity and the types of proof required. We give a simple denotation: let $\mathbf{N}_V = \{1, 2, ..., n_v\}$, where n_v is the number of integrity levels. Let $\mathcal{P}$ be a set of types of proof. Now define a valuation $v_{\mathcal{P}} : \mathcal{P} \to \mathbf{N}_V$ by: for each $p \in \mathcal{P}$, let $v_{\mathcal{P}} = n$, where p has been (informally) assigned integrity level n. In practice, a table may be drawn up listing for each level of integrity the designated proof measures.

In standard guidelines on methods required for the development of safety-related software, only those functions which are assigned the highest integrity level stipulate the use of formal methods, but the formality of measures specified is a minimum, and where feasible, in terms of e.g. efficiency, designers should consider making use of the

potential extra benefit offered by the scope and correctness of formal analysis.

Designation of Safety-related Systems

This phase is the output of the safety requirements specification phase, specifying the parts of the system that are safety-related, assigning to these some indication of the relative significance and hence the safety integrity levels. At this stage it may be decided what are to be the methods of control that are to achieve the required levels of integrity for each of the designated systems.

This marks a change of emphasis from systems requirements analysis to systems design. In particular, in the formal setting, the focus is now on the task of building a model. The methods of control are design strategies which are intended to fulfil the requirements. This is the context in which are set the verification and validation issues raised in the discussion of formal methods (Chapter 2, section 2.6). For instance, we could take advantage of the fact that the model can be built in modular fashion such that in many cases if some property holds for a component, then it holds for the entire system.

Methods of Control

Once the requirements have been determined for the risk reduction, methods of control are specified for the system to effect this reduction. FMECA may be used to establish possible failures of the system, for which further methods of control may be specified to ensure that *its* integrity is assured.

As for safety integrity levels, we provide a simple means for denoting the relevant relationships as can be applied to formalised objects, this time between hazards and methods of control. We may take the view that we select from a universe of informally specified methods of control, $\mathcal{M}$, say, which, in reality, is likely to be an infinite set. As above let $\mathcal{H}$ be a set of denoted hazards. We define a relation $\Upsilon \subseteq \mathcal{M} \times \mathcal{H}$ by $(m, h) \in \Upsilon \iff$ "m is a method of control for h". In practice, once again, this set is chosen informally. Also note that methods of control will usually have to address requirements other than those related to hazards.

In our case study, one task is to use formal methods to validate an informal document that may be considered as a proposed implementation of a set of user requirements. Any such document may be modelled as a set $\mathcal{D}$ of items d, such as paragraphs, tables, and their substrings/subtables. We may then regard $\mathcal{D}$ as containing a subset $\mathcal{M}_D \subseteq \mathcal{M}$ of such methods specified informally.

Just as there may or may not exist a formalisation of hazards, it may or may not be the case that the document contains an appropriate method of control for a given hazard. We therefore define a valuation $v_{H_D} : \mathcal{H} \rightarrow \{\texttt{tt}, \texttt{ff}\}$ by: for each $h \in \mathcal{H}$, let $v_{H_D}(h) = \texttt{tt}$ if there exists a method of control $m \in \mathcal{M}_D$ such that $(m, h) \in \Upsilon$. Let $\mathcal{D}^+ = \{d \in \mathcal{D}\}$ such that $v_D(d) = \texttt{tt}$. Let $\mathcal{H}_\mathcal{D} = \{h \in \mathcal{H}$ such that $v_D(h) = \texttt{tt}\}$, then $\mathcal{H}_\mathcal{D}$ is a subset of $\mathcal{H}^+$. As proposed standards documents may be placed in this class, valuations of this kind, using formal techniques, can help to assess their feasibility and contribute to their enhancement.

3.5.4 Design, verification and validation

The lifecycle model then conceives the process of designing the appropriate system in terms of the following:

Validation Planning, Design & Implementation, Verification, Safety Validation

The basic concepts and principles for these stages are well established (summarised in Appendix A of [B. 89] and are discussed in detail in, e.g., [Wic92]). It is realised that procedures will depend upon the particular system and context being developed.

The impression given by the standard lifecycle model is that validation is performed after any number of incremented designs: note the two way arrows between 'Design and Implementation' and 'Verification'. This appears restrictive, so we propose that the formal development be carried out as a sequence of stages, as conceived using the model given previously in Figure 2.3 which is specially designed for the formal development. Its structure provides complete coverage of the stages indicated in the lifecycle, whilst emphasising the nature of flow in step-by-step refinement.

3.6 Managing the refinement

In this section we look at the support required for managing the development of formal software items. In large projects such as presented in the LOTOSphere reports, many issues are discussed, ranging from systems design to technical points; however, the issues surrounding the management of change are not so clear. In response, the next sections are intended to clarify matters through the discussion of CM and Risk Management frameworks.

One of the few papers that focuses specifically on making changes to formal items is [BW94], in which there are some general guidelines for maintenance, including an analysis of the types of changes that can be made. The paper establishes this approach within

the context of CM control. We review here the reasons for change in general and discuss them from the perspective of refinement, with some reference to the LOTOSphere Design Methodology [LOT92c, LOT92b]. In Bustard and Winstanley's work, the requirements for change are partitioned into three steps:

1. understanding the need for change;
2. implementing the change;
3. evaluating the change.

The requirements analysis stage establishes the main requirements of the intended system, often to quite some detail. Owing to the complexity of all but the simplest systems, it is generally impractical to incorporate all these requirements in an initial model. Hence, a development in stages is used, so a succession of changes will be required, thereby needing the repeated application of the steps above. Understanding the need for change will be largely determined by the extent to which the requirements have been implemented.

Building the system may also require alternative *design trajectories* (development paths). In the detailed methodology guidelines for LOTOS processes, as presented in the LOTOSphere Design Methodology, a design trajectory is considered as a *tree* of refinements, rather than a single path. Figure 2.3, which shows just one completed path, should be seen in such a wider view: we discuss the suitabilty of a tree as a model in section 3.6.2.2.

Where more than one path for the refinement is being investigated for suitability, evaluations are performed at the completion of each item, with respect to consistency and safety integrity from the ongoing safety analysis and requirements derivation. This is characteristic of a *tree search*: a suitable node is 'found' when an item is deemed OK, thus enabling undesirable or inconclusive branches to be 'pruned'. Feedback from 'failed' branches is useful as input to the other branches being investigated.

In succeeding sections we discuss each of the steps with respect to our context, starting off with the mechanics of change, i.e. implemention.

3.6.1 Implementing Change as Formal Transformation

In the formal setting, change that is given a well-defined meaning is termed *transformation*. In any transformation, there are many issues to consider, e.g., once some property of a specification has been proved, does this property still hold after a given modification? In the formal context, one has a range of options: one can prove *a priori* that according to a given notion of correctness, certain changes preserve certain relations and

properties. These are called *Correctness Preserving Transformations* (CPTs) and for LOTOS there is a list given in [LOT92a], some of which are supported by tools. On the other hand, one can use methods of proof to show *post hoc* that properties hold, but this usually takes more effort. Ideally, we would like the whole process of refinement to be a sequence of CPTs.

In summary, implementing changes to formal items in the refinement can be done in various ways:

- according to a predefined transformation (or template) where semantic effects are already known
- according to a series of transformations on the model, whose effects are not known
- according to a series of manual changes

The first requires evaluation to the extent that the updated model can be analysed to provide further feedback into the requirements: in general, a complete picture of the effects of a given transformation may not be fully known, even though some relations embody the notion of "not introducing any undesired behaviour". The second and third, which provide increasing levels of uncertainty, need verification and validation after the changes have been made.

The notions of correctness above are for any item and for any relation, whose strength will have already been defined by the valuations for 'methods of proof'. The kind (or, preferably, choice) of transformations between respective items needs to take account of design practices, which themselves need some management framework. Such a framework is discussed next.

3.6.2 Configuration Management

In this section we provide a framework which enables a closer examination of requirements for change, the kinds of relationships that can exist between formal objects, and ways of evaluating the change with respect to the nature of these relationships.

For instance, one can apply principles of CM to formal structures to facilitate the construction of useful CPTs. One aspect is to establish 'loose coupling' to minimise dependencies after which it should become clearer how CPTs can be designed to preserve certain item properties and aid in the construction of methods of control.

3.6.2.1 Items, Configurations and Configuration Graphs

Two concepts that are essential to CM (as provided in, e.g., [Whi91], which is a useful guide to the subject) are defined as follows:

> *item* Software which is treated as a unit for the purposes of CM.
>
> *baseline* An item which has been fully approved serves as a basis for further development and can only be changed through formal change control procedures. Usually the term baseline refers to all items produced by a phase of the project lifecycle.
>
> [IEEE-729 Standard Document, 1983]

One type of 'super' item is a *configuration*, being the collection of items which fulfil a particular purpose, such as a safety case. A configuration is in general a representation of the entire system under construction, being a snapshot of the collection of items which make up a stage, covering various development activities. The items with which we are concerned start off as formal, so formal relations should be established between them. The whole process of their refinement can become unambiguous if all these items can be set in a suitable formally defined framework. Further, formally modelling *all* items in the management proces may aid understanding of the overall development.

A general structure that provides a suitable model is that of a *directed graph*, whose nodes represent configurations and whose edges represent changes in the configurations. Whenever alternative changes are made to a given configuration, then more than one edge is produced from the respective node. The items may be related with respect to their relative development in time, inducing a simple partial order. Each node may be labelled with a version number – according to the configuration's level (in terms of the depth of refinement) and the particular branch.

Such a graph model, as illustrated in Figure 3.2 is generic, which may be variously instantiated for the development of component structures. Note, for instance, that configurations can consist of configurations (e.g., the overall CM can contain as one item a collection of independently evolving partial subcomponents...)! For small systems, it is usually most convenient to think in terms of just one CM model; but for very large activities we may apply the principle of abstraction and so several may be suitable.

Referring to the figure, a *configuration graph* (or *CM graph*) CG is defined as a labelled graph (labelled unambiguously) where each node (item) is a configuration, belonging to a set $\mathcal{I}$ of items; a set of relationships $\rho \subseteq \mathcal{I} \times \mathcal{I}$ may then be defined between items. A label l is associated with each node to denote the *version* of the configuration, where

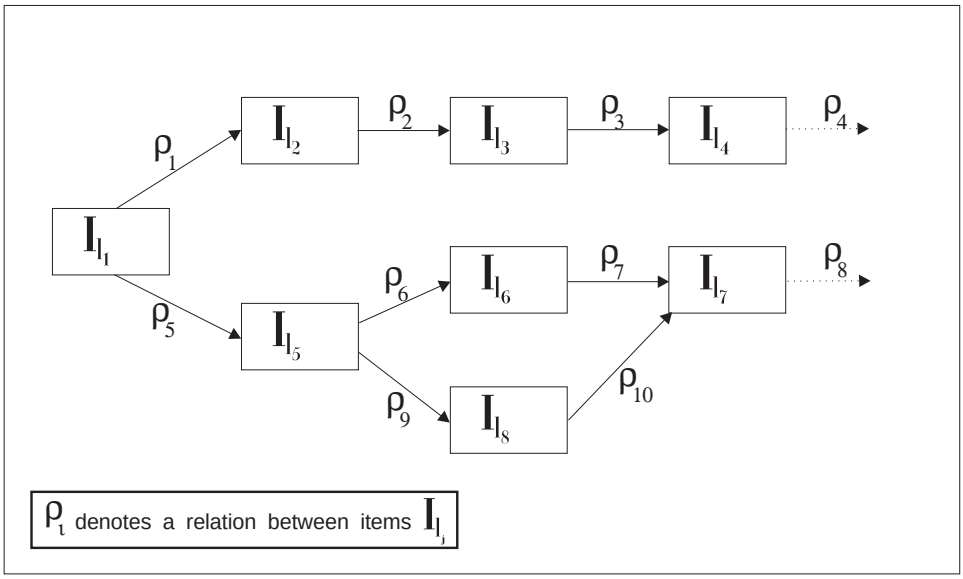

Figure 3.2: A generic Graph of refinement in CM

versions are defined according to some algorithm that provides an unambiguous labelling scheme that allows for branching off any given branch: one such scheme is used in RCS [Tic85], a version management tool. This a straightforward inclusion, which supports the view that modelling the overall CM process in terms of a directed graph is a fairly faithful representation of established industrial practice.

In order to reason about configurations, we make use of the fact that they consist of sets of items of various types. Hence, certain *intra-configuration* relations (item dependencies) may be sought to check especially for internal consistency. Further, *inter-configuration* relations may be established between items in different configurations: analagous to the methodology for hazards, 'horizontal' relations may be established to compare alternative models or other items along different branches, and 'vertical' relations may be established to compare items along the same branch. The nature and scope of the relations will depend upon the relative types of the items being compared.

3.6.2.2 Strands within CM

Since configurations contain all manner of items, it is usually not possible to define a formal relationship between them except for the purpose of modelling management processes. However, by analysing constituent *strands* (or sequences of successive items), through an instantiation of the graph model, the formal relationships can be drawn out. Hence, where the items are models or formulated requirements, we may investigate these

relationships, defining semantics for ρ, according to the kinds of transformations that may be performed. In particular, if each ρ_j is a preorder contained in a relation ρ_X, say, then we have the following:

Let I_0 be an initial model, and $I_1, I_2, ..., I_n$ be a succession of refinements such that

$$(I_0, I_1) \in \rho_1, (I_1, I_2) \in \rho_2, ..., (I_{n-1}, I_n) \in \rho_n$$

and where $\forall j.\rho_j \subseteq \rho_X$. Then (by induction) we have that $(I_0, I_n) \in \rho_X$.

If ρ represents the notion "is an implementation of" then this implies that I_n is a (valid) implementation of the specification I_0. We may also say that each ρ_j *is of type* ρ_X.

Regarding the strands of items – through which are defined the vertical inter-configuration relationships – the structure of their evolution needs to take account of theory and practice related to the relevant activities. Specifically, regarding the development of formal models, from a theoretical perspective, the development needs to be undertaken in a sound and, preferably, complete manner; important practical constraints include allowing a team of designers to work concurrently on different aspects of the same problem, implying the construction in parallel of several models whose composition is intended to produce an overall model of the system. In this instance, a directed graph is a sufficient representation since parallel activities may be modelled as branching from an initial 'empty' node and edges can be joined later on as specifications are merged. This is not so for a tree representation.

For large systems in general, various techniques have been employed to handle complexity, based upon some paradigm, of which *object oriented* approaches are a popular example [Boo91]. Another paradigm that has been developed for distributed systems is that of a *viewpoint* [FKN$^+$92]. This has subsequently been adopted for emerging standards on Open Distributed Processing [1-495]. Note that the LOTOSphere Design Methodology, bearing close comparison with standard waterfall models, does not cater explicitly for such parallel development.

Viewpoints are readily incorporated as concurrent strands in the directed graph; key issues that then arise concern *consistency* between items in different viewpoints and their merging (or *composition*) into one item with a single viewpoint. Promising work on tackling these issues for system models has been treated in [SBD95], where this approach is addressed within the context of refinement of LOTOS specifications. The paper gives the following notion of consistency for (partial specifications) which are intended to be composed further along the development path.

Definition (Consistency and Composition)

Given two specifications S_1, S_2 and a refinement relation $\mathbf{ref} \subseteq \mathcal{SPEC} \times \mathcal{SPEC}$, S_1 is *consistent* with S_2 with respect to $\mathbf{ref}$, denoted $S_1\ C_{\mathbf{ref}}\ S_2$ iff $\exists S \in \mathcal{SPEC} \cdot S\ \mathbf{ref}\ S_1$ and $S\ \mathbf{ref}\ S_2$; any such S is called a *composition* of S_1 and S_2. ($\mathcal{SPEC}$ denotes the set of all specifications).

This definition forms the basis of a systematic treatment of various notions of refinement and the definition of their respective notions of consistency together with some results on composition according to various refinement relations and the definition of various composition operators. We are able to generalise this definition both to fit our wider framework and, specifically, to allow different viewpoints to have different notions of refinement. Hence we have the following:

Definition (2-relation Consistency and Composition)

Given two specifications S_1, S_2 and refinement relations $\mathbf{ref}_1, \mathbf{ref}_2 \subseteq \mathcal{SPEC} \times \mathcal{SPEC}$, S_1 is *consistent* with S_2 *with respect to* $\mathbf{ref}_1, \mathbf{ref}_2$, denoted $S_1\ C_{(\mathbf{ref}_1, \mathbf{ref}_2)}\ S_2$ iff $\exists S \in \mathcal{SPEC} \cdot S\ \mathbf{ref}_1\ S_1$ and $S\ \mathbf{ref}_2\ S_2$; any such S is called a *2-relation composition* of S_1 and S_2.

In using process algebras for system modelling, specifically LOTOS, such relations are based on labelled transition systems, which, as has been detailed in the previous chapter, offer much scope. There has also been work done within the viewpoints setting to relate models in different languages, specifically LOTOS and Z [DBBS96]. These papers demonstrate how models that evolve in parallel may be related. In the next chapter we treat an alternative problem of the relationships between concurrently evolving *requirements specifications* and system models as befits the focus on safety.

3.6.2.3 Target baselines for CM

Once safety integrity levels and then methods of control have been determined, we can plan the phases of the project to reflect the various requirements, thereby providing some focus for the change management. Although the requirements may have to be modified as the model develops, it is helpful to have a series of targets to aim for, in terms of the amount of requirements accounted for in the evolving design. In view of the three perspectives in making changes, we introduce the notion of *target baselines*, being a set of requirements on items and which is associated with a stage in development.

In determining such targets, we note that system development may increment baselines according to:

- mission requirements: i.e. modify the functionality
- safety requirements: i.e., introduce or modify some method of control
- other non-functional requirements (such as performance)

For safety-related systems, risks should be seen to help drive the construction process. Using the characterisation of hazards or causes as *faults*, we can draw up a sub-classification which lists the different ways we can increment baselines with respect to the safety requirements:

- fault inclusion
- fault prevention (method of control)
- fault tolerance (method of control)
- ... *etc.*

Similar subclassifications could be introduced for other kinds of requirements.

Applying the 3-step paradigm for change, one must consider when moving from one baseline to the next what impact do certain changes have on the system's integrity – the relationships between components especially; and how do they affect the overall safety?

3.6.2.4 Recording Changes in the Refinement's CM

At each refinement step, we need to determine for each item how it is affected under the change. As our refinement contains multiple steps, this entails a lot of checks. So in order to record this process, a *change history* needs to be maintained, with special attention to the activities of verification and validation. For each item, we may set up a change history to document changes between item versions.

Contingent with the change history is the notion of item 'status', being some level of approval for the item. The software items to which CM has been traditionally applied are non-formal — source code, object files, ... etc. The kind of status these have are typically to do with testing, with simple stamps for whether the item works okay by itself and perhaps within a larger system. Even if such tests are carried out in accordance with 'best practice' or standards, they can still be prone to subjective interpretation (and hence variation) between companies, and what they record cannot be proved mathematically.

Formal items are unambiguous and will require records where the status of items will be with respect to tests like the above, but also to the satisfaction of universally

defined formal relations. We envisage approval based on the extent to which an item has satisfied consistency and properties fitting the levels of verification and validation stipulated according to integrity levels. There is a close relationship between the management of change and that for risk: the properties of a specification item should correlate with methods of control of hazards, and the status of an item will depend on the extent to which it achieves the respective integrity levels.

Although the change history can record the status in the development of individual items, it becomes inadequate in recording the wider status of a configuration, so other means are required. As a solution, information regarding properties and their preservation can be recorded in the Risk Management Log, whilst wider issues of integrity can be recorded in some other log, perhaps we could call *Consistency Log*.

The Consistency Log could record the integrity of a configuration in terms of its component items, with a list of the relations that must be satisfied between them for the configuration to have an 'approved' status. Each such item would be placed under version control and then versions can be selected for the verification activities. This is an open topic, which should be the subject of future research.

3.6.2.5 Tool Support

Tool support for software CM is widely available, but generally without any facilities for supporting formal items; most cater for standard programming languages such as `C`. Hence, more of the workload falls on the tools that have been explicitly developed for formal methods analysis and crude tools that are able to handle any items.

For instance, regarding LOTOS, one can expect development to be carried out under LITE (Lotos Integrated Tool Environment) [PvEE92] which runs under the UNIX and X windows system and operates on most SUN system. It is integrated in that all tools share a common representation (CR) and have been developed to support the LOTOSPHERE design methodology [LOT92b, LOT92c], which is based on step-wise refinement where design decisions can be gradually incorporated at each step. LITE supports many aspects including the creation of specifications with a structure editor which checks syntax and semantics, a flexible simulator SMILE[EW93] and other tools for the generation of trees, G-LOTOS graphical representations, and even a report generator. As described in Chapter 2, other tools such as the CADP toolset [FGM+92] offer more powerful *verification* facilities.

However, these provide generally little assistance for CM, so a combination of tools are used for maintenance tasks. Thus, specifications may be built up by use of a

template which is pre-processed using some macro processor to generate a target LOTOS specification. Such a template may 'include' components (as for `C` source code), e.g. a data type's definition, a buffer definition, etc. Versions may be maintained by using RCS. Tools are also available to facilitate the building of configurations, for which the one general tool is `Make`[Fel79]. There is also some support in LITE: for instance, a simulation option invokes a makefile which checks syntax, semantics and translates a LOTOS text specification into a CR for use in SMILE.

3.6.3 Risk Management

Providing safety integrity comes from risk management ('RM' for short), for which keeping a sensible record is part of the process. All this information can be maintained in a *Risk Management Summary* (or *RM log*), which is standard practice for giving a written account of risk. Such practices are generally required in any software project management. In our particular safety-critical context, we consider the RM Summary used for PEMS as discussed in [Bib95] and given below (Table 3.1).

Baseline $< n >$ Branch: $< a_1, a_2, ... >$							
Hazard No.	Baseline Entered	Hazard & Cause	Risk	Requirement [Method of	Requirement Reference	Verification & Validation	RFU
				Control]			

Table 3.1: A Risk Management Log

We conceive the RM Log as consisting of a *tree* of logs, corresponding to the tree of baselines, so the logs are updated each time there is a baseline increment along the branch. At any given moment, there should be 'in circulation' a set of logs such that all branches in the development are accounted for. The remaining (previous) logs should be kept for future reference. Hence it would be useful if the RM Log itself should be placed under version control. The format of individual logs, which correspond to nodes in the baseline tree, reflects safety, the use of formal methods, and the use of CM management framework.

In general, developing a record for risks for non-formal items is fairly well understood. However, there appears to be little material that even asks whether or not formal items may need special requirements. In response, we firstly assert that formal items more than any other require traceability, otherwise proofs cannot be justified. Hence, we consider here the RM Log's feasibilty for supporting formal models in this way, though only

briefly. Much depends on what tool support can be provided; in particular, the use of hyperlinks could open up the use of graphical representations such as SSG's, thereby aiding traceability.

Notes and Recommendations regarding the Risk Management Log

- The entries under `hazard & cause` should consist precisely of all those hazards identified in the hazard analyses as having a bearing at baseline depth n. In our formal context, we start with an abstract system in which properties may be deemed to hold 'vacuously' until explicitly mentioned provided that there is some record kept that indicates the level of (in)completeness. The measures for these can be expressed in terms reflecting the definitions in Section 3.5.2.1.

  As the refinement proceeds, whenever additional hazards are revealed they should be added to the log. They will typically become finer in level of granularity.

- The `Baseline entered` is to provide a record of when hazards were identified, in terms of the stage of development (baseline depth) – '0' denotes those hazards determined in the initial hazard analysis. The aim of this is to give insights into the kinds of problems which can 'crop up' unexpectedly, and at what stage.

- This column gives summary descriptions. More than one `Requirement` may be elicited for a single hazard, with the option of making explicit a `Method of Control` – which can be omitted when the overall design is expected to fulfil the requirement. Note also that the same `Method of Control` may be employed for several hazards, but the verification and validation need relate only to the 'Cause of Hazard'.

- A particularly important feature of the log in our development is the need to ensure that properties relating to `verification & validation` continue to hold during subsequent changes. Consider, for instance, that a hazard is identified in baseline 1, when a corresponding property ϕ is formulated that a model S_1, say, must satisfy. Then we require *validation* of $S_1 \models \phi$. Suppose that for subsequent baselines, we have a succession of models S_2, S_3, ... Then we would validate that ϕ continues to hold through *verification* of $S_1 \rho_1 S_2$, $S_2 \rho_2 S_3$ and so on for suitable relations $\rho_1, \rho_2, ...$ Thus, the validation and/or verification which justifies the integrity of the `method of control` should be given special attention; this column should contain:

  – a (detailed) natural language statement of a desired property

  – the formalisation of the property

- description of steps in the formal proofs, with references to Lemmas and Theorems used (usually given in appendices or other documentation such as technical reports)
- Some verification may be dependent upon previous results. If so, these dependencies (which may be of several types) must be stated clearly.

This column can be conceived of as a matrix in itself with the number of columns corresponding to the number of formal justifications required. Ideally, methods of control should always be proposed hand-in-hand with a plan for verification. The introduction of a method of control may be considered as some transformation. Depending on whether or not it is a sequence (or composition) of CPTs, its verified status may or may not continue to hold; if not then some proof is required and the risk management logs should be updated, indicating the extra proof required under the change. This applies for all subsequent changes. The outcome of this in terms of the risk management log is a matrix of measures.

3.7 Observations and Conclusions

In this chapter, we have re-capitulated the general impression that the uptake of formal methods in industry is poor. In trying to discover some reasons for this, we have identified a number of weaknesses on the part of the formal approach, most of which stem essentially from the lack of integration into wider software development practices.

In response, we have provided a safety-oriented framework based on systematic consideration of the amenability to formalisation of each of the stages in the standard Safety Lifecycle Model. This has raised many open issues, many of which will have to be analysed in greater depth. Here, our main concern is changing the shape of the development in such a way that the formal cornerstone of proof is made widely useful: a consequence of our activities is that to reflect the particular needs of the refinement, the tasks of verification and validation have been given more attention.

In so doing, the process has drawn attention to the fundamental issues of consistency and completeness. Producing safety-critical systems requires a number of stages, each of which ought to satisfy some form of completeness. It appears that in order to ensure ultra high integrity, there needs to be a systematic treatment of completeness at each step. In the face of such a situation, the levels of dependability that have apparently been achieved without formal methods is remarkable.

This has been particularly evident in the treatment of basic safety concepts such as hazards. A broad framework has been defined that provides some measure in the ability of individual formal languagese to capture notions of hazards and their relationships. From this perspective, it is clear that for a language to model requirements completely is a considerable task. However, consider that embedded systems are built using electronic components, whose behaviour certainly *can* be modelled formally. If there is confidence in the completeness of such embedded systems (dependent on whatever), then this indicates the potential completeness of a formalisation. In traditional engineering, confidence has often come from experience. However, as the dependence on software increases and systems increase in novelty and complexity, completeness will have to be more part of the design from conception, for which formal methods seems the most likely candidate.

Some of the special emphases and modifications are summarised in the lifecycle model given on the lefthand side in Figure 3.3, which splits the development process into three major phases with internal stages (or sub-phases). For this model, we have taken each stage and established the relationship with the formal development, given as the dotted arrows from left to right: the thicker arrows indicate a dependency, whilst the thinner arrows indicate a weaker relation of acting as guidelines.

The model that is presented in Figure 2.3 assumes a stable environment, in which once requirements are determined, they remain fixed for that stage in the development and any modifications, additions or refinement happen further into the development. A more general and realistic model would provide for corrective changes to requirements established previously. These changes may be identified with respect to the stage in development by reference to the RM Log, specifically the baseline and branch; new CM items can be created accordingly. The extent of the impact will depend on the location of the divergence from latest requirements. Any change to requirements, just like an error in initial design, has to be 'rippled through' the stepwise designs from the point of divergence. If the nature of the proposed system is such that it is prone to frequent major changes in requirements, then the system may be ill-conceived.

Coping with these changes is difficult, so this makes it essential that user requirements are established, understood and agreed upon as soon as possible. To this end, Soft Systems Analysis [Che81], preferably with formal support as being researched presently [BL95], may be useful to ensure that what the customer needs is clear. However, for some systems the factors which influence the safety integrity levels are very much subject to change depending upon the external world. Providing high integrity for these systems is an even tougher challenge.

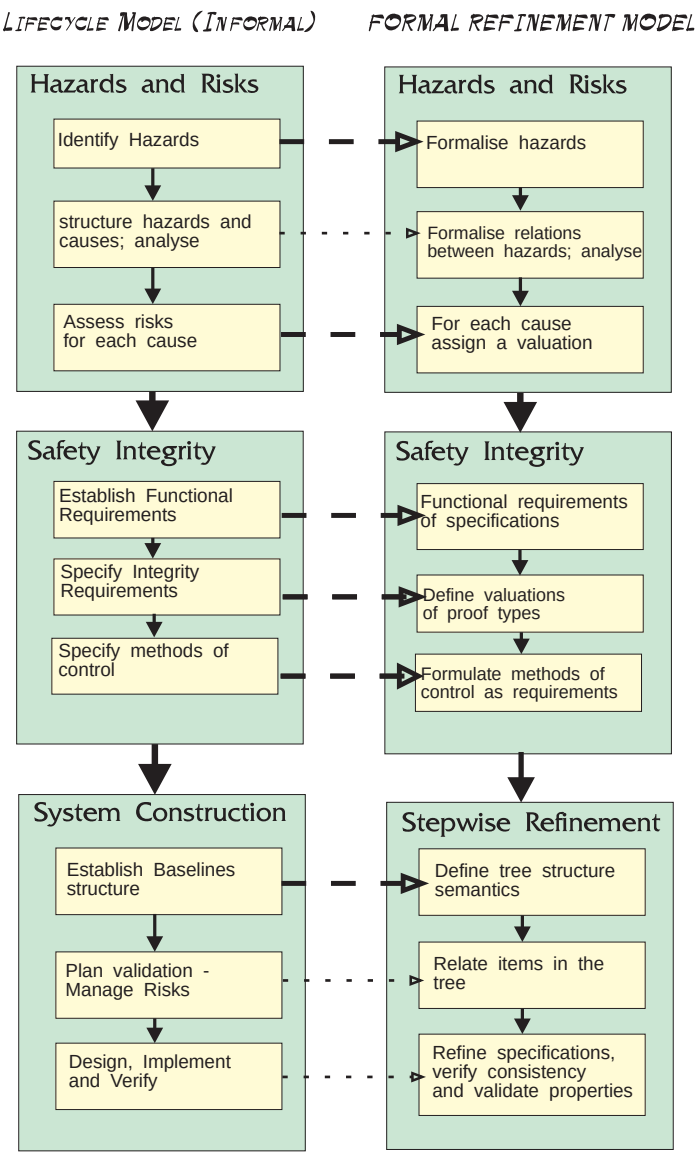

Figure 3.3: The Lifecycle Model and its bearing on the formal refinement