Fundamental Concepts for Fault Tolerant Systems

J. R. Garnsworthy

1990

Submitted for the degree of PhD in the Faculty of Science

Abstract

In order to be able to think clearly about any subject we need precise definitions of its basic terminology and concepts. If one reads the literature describing fault tolerant computing there is less agreement on fundamental models, concepts and terminology than would perhaps be expected. There are well established usages in particular subcommunities and many other individual workers take care to use terms carefully. Unfortunately there are also many papers in which terms are freely applied to concepts in an arbitrary and inconsistent way.

This thesis attempts to bring together some of the concepts of fault tolerant computing and place them in a formal framework. The approach taken is to develop formal models of system structure and behaviour, and to define the basic concepts and terminology in terms of those models. The model of system structure is based on directed graphs and the model of behaviour is based on trace theory.

Contents

List of Figures

Chapter 1

Introduction

In order to be able to think clearly about any subject we need precise definitions of its basic terminology and concepts. If one reads the literature describing fault tolerant computing there is less agreement on fundamental models, concepts and terminology than would perhaps be expected. There are well established usages in particular subcommunities, such as at Newcastle University[21] or UCLA[4], and many individual workers take care to use terms carefully. Unfortunately there are many other papers in which different terms are freely applied to concepts in an arbitrary and inconsistent way.

At the tenth IEEE Conference on Fault Tolerant Computing (FTCS-10), in Japan, a group of people involved in fault tolerant computing recognised the problem after discovering in many of the papers inconsistencies in concepts and terminology. So the following

year a joint session of IFIP WG10.4 "Reliable Computing and Fault Tolerance" and the IEEE Computer Society Technical Committee on Fault Tolerant Computing was held just before FTCS-11 in Portland, Maine. The session was devoted to presentations on concepts and terminology, and discussion of the problem.

At FTCS-11, D. E. Morgan was requested to organize a joint subcommittee of the two groups and this sub-committee, "Joint Subcommittee on Models, Fundamental Concepts and Terminology", had its first meeting in September 1981. This group met over the next two years, including a panel session at FTCS-12, to discuss the problem of the lack of definitive terms for fault tolerant computing. The session was organised and chaired by Morgan and papers were presented by T. Anderson[2], A. Avizienis[4], J. C. Laprie[19], H. Kopetz[17] and A. S. Robinson[23].

The work in this thesis grew out of their definitions, more specifically those of Anderson and Lee[2,3], Laprie[19,18] and, in addition to those represented on the panel, Cristian[10].

The approach taken in this thesis is to produce a formal model of system structure and behaviour, and to define the basic concepts and terminology in terms of that model. We are unable to produce a single unified model and instead produce two separate models, one for system structure and one for system behaviour. However, we are able to deal with some other important concepts which are not specific to fault tolerant computing, namely the two related concepts: atomicity of action and behavioral abstraction.

This thesis develops a model of structure using the theory of directed graphs, based on the work of Laprie[18], Best[6,7] and Parnas[22]. The idea behind the model is that structure is a relationship between components of the system and that there is a basic relationship, 'is a component of', which is meaningful for all systems. That is, all systems are composed of components, which themselves are composed of components and so on[21]. This gives us a tree based organisation for our systems.

Further, other relationships between components, for example 'uses', are orthogonal to 'is a component of'. This is because other relationships which describe how we see the system to be structured, only have meaning between components that are not contained within one another. For example, a factory does not 'use' a machine. A machine 'is a component of' a factory, a worker 'uses' a machine. This is based on Laprie's Layer-Component graphs[18] and is discussed in detail in Chapter 3.

Our model defines the idea of a 'level of detail' as a cut through the tree based on 'is a component of', and lets us consider the orthogonal relations at any level of detail. The model is based on Best's structured occurrence graphs[7]. These 'levels of detail' are ordered in a lattice with the entire system at the top and the maximum decomposition at the bottom. This means that the 'highest' level of detail contains the least amount of detail and vice versa. We call our model "structured graphs".

This thesis applies structured graphs to the problem of defining the concept of atomicity of action. This is where we view a group of actions as having the same properties as a

'primitive' action, i.e. such a group of actions can sensibly be viewed as being indivisible. We define a structuring relationship ('is a component of') for events, and a relationship 'immediately succeeds' which is orthogonal to this structuring relationship. This defines a structure graph and we argue that the presence of loops in a level of detail of that graph means that some action is non-atomic and then go on to argue that a group of actions are atomic if nothing outside that group 'interferes' with the action, i.e. occurs strictly after one part of the action and strictly before another. We compare our formalisation with a previous one by Best[7].

We develop a model of system behaviour based on the theory of traces[8,14]. We do not equate the specification with the system, as is common in such approaches to modelling system behaviour. This would make it impossible to define the concept of failure. In the model the system interacts with its environment by the passage of objects through its interface. The interface is the dividing line between the system and its environment.

Three different concepts of abstraction which can be applied to the interface of a system are considered in terms of the model, the first two are taken from the work of Brookes, Hoare and Roscoe[8].

The first of the abstraction concepts we deal with is 'hiding'. This is a process where we ignore the parts of the system interface which are irrelevant to the problem under consideration. For example, when considering the behaviour of our program on a time-sharing computer we normally only consider the behaviour of the computer at our terminal

and 'ignore' what is occuring on terminals other than our own.

The second concept is 'resorting'. Here we change our interpretation of the objects passing through the interface. For example, one can consider the signals passing through a terminal port on a computer as purely electrical signals, as bit patterns, as numbers or as characters.

The final concept is 'collapsing'. This process allows us to aggregate parts of the interface into a single part. For example, we take 8 one-bit channels and consider them together as a single channel through which passes an integer in the range -128 to +127. Formally, resorting is a special case of collapsing where we collapse a single part of the interface to a new single part of the interface. The concept of collapsing is closely related to the concept of atomicity and we discuss this relationship.

State is considered and we argue that the notion of state is actually (and perhaps surprisingly) an external phenomenon. That is, to predict the state of the system (or of a component) we require no knowledge as to the structure of that system (or component). State is the changing part of a model which tells us what the system is going to do next. The model could have come from observation, knowledge of how the system is constructed or knowledge of how the system should behave. It can be seen from this that the state of a system is defined *with respect to* a model of that system's behaviour.

The way in which the state of the system is represented is not fixed. A representation

is chosen as best suits the use to which our model of state is being put. For example, a programmer debugging a program has a different view of the programs state than a user would. The programmer would view its state in a way which related to the internal structure of the program. The user would view the state in terms of the domain application.

We call this view of state the 'external state' and define the minimal set of states to be any set isomorphic to the set of future behaviours. We go on to define the 'internal state' of a system at a level of detail to be the ordered set of external states of its components given by that level of detail. Thus, the internal state at the 'highest' level of detail, that is the one containing the lowest amount of detail and so representing the system as a whole, is the same as the external state of the system.

Based on our concept of state we describe what we mean by programming a system and this leads us to the concepts of a program and of an interpreter. We discuss interpreters and interpretive extensions.

We define failure as deviation of system behaviour from that defined by a specification. That is, a system fails *with respect to* a specification. If two people differ in their opinion about whether a system has failed, that is, one says the system has failed and the other says it hasn't, then they must be working with different specifications. They may, of course, arrive at these different specifications through different interpretations of the same document. Also, we use the concept of failing with respect to a specification to

construct a hierarchy of failure modes based upon a set of specifications[19]. We use this to consider graceful degradation and compensation.

We say that the state of a system is erroneous if it is different from the 'correct' state and the system may fail at some future time, after a sequence of valid interactions with the environment. Given our definition of state, this will require two specifications, one describing the 'required' system behaviour and one defining the 'actual' system behaviour. The second of these specifications is acting as a model of the systems actual behaviour. The first is a specification of the systems required behaviour. Two specifications provide us with two states, since states are defined with respect to specifications. One is the 'actual' and erroneous state, that is the state (we believe) the system is in. The other is the 'correct' state, the state the system should have been in, had it functioned correctly. We define an error as the difference between the erroneous state and the correct state. How we decide on the difference between two states is dependent on the representation we have chosen for those two states.

Hence, errors are defined *with respect to* two specifications.

We discuss error propagation and error recovery. Because we have two separate models of behaviour and structure rather than a single unified one, we are unable to be as precise as would have been desirable.

We describe what a failure is and what an error is and argue that these are both external

phenomena, in the sense that we have to have no knowledge of the internal structure of a system to define them, recognise them or locate them. We next look at the cause of these phenomena. We define a fault to be the possible cause of a failure, following Laprie[18]. We discuss fault avoidance, that is how not to create faults, fault removal, that is how to identify and remove faults, and fault tolerance, that is how to stop an existing fault actually causing a failure.

This thesis is structured into six chapters; the first is this introduction. The second is a literature survey discussing the work which has influenced the contents of this thesis and putting the thesis into historical perspective. It also compares and contrasts a number of definitions for reliability terminology given by different people.

The third chapter looks at structure and defines our model of system structure, structured graphs, based on the theory of directed graphs[9]. It also applies this model to the concept of atomicity of action. This chapter is an improved and corrected version of the paper awarded the Phillip Merlin Memorial Prize for 1984[12].

In the fourth chapter we consider a model of system behaviour, based on the theory of traces which underlies Hoare's CSP[8,14]. Using this model we consider behavioral abstraction and the concept of state.

Chapter five considers a number of reliability concepts. In particular it gives precise definitions for failure, error and fault. The final part of this chapter examines how our

definitions relate to those of others described in chapter two.

Finally, in chapter six we draw together the work we have done and consider how it could be improved and extended.

Chapter 2

Related Work

In this chapter we consider related work which has defined terminology and concepts in the area of fault tolerance and reliability.

2.1 Melliar-Smith

The definitions given by Melliar-Smith and Randell[21] are in many ways the closest to ours and we can trace our approach back to them. Systems are defined as being made up of components (which are themselves systems) related in such a way as to provide a defined service. The internal state of a system is defined as "the aggregation of the external states of its components." The external state of a system is defined to be "the

result of the conceptual abstraction function applied to its internal state." Though we make use of the same definition of internal state, we take a more abstract approach to the definition of state. However, there appear to be two deficiencies with the approach to state taken. The first is seen in the following: "During a transition from one external state to another external state, the system may pass through a number of internal states for which the abstraction function, and hence the external state, are not defined." This implies that the system can exist without being in any particular external state. We do not accept this. The second problem with the approach to state is the common one of confusing the state of the system and the output of the system. We will return to this in the next section.

A failure is defined in what can be considered to be the standard way as: "a failure of a system occurs when that system does not perform its service in the manner specified." The definition of error accepts, as we do, the subjective nature of identifying erroneous states. "We term an internal state of a system an erroneous state when that state is such that there exists circumstances (within the specification of use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault." This differs in two ways from our approach. Firstly, with our more precise view of state we can define an erroneous external state and, secondly, we do not need to introduce the concept of fault to define an erroneous state.

A fault is defined as the cause of an erroneous state and a potential fault is a construction

within the system that may cause the system to enter an erroneous state. We would argue that this is not a usefule distinction to make. The only difference between these definitions is that a fault has already caused the system to enter an erroneous state whereas a potential fault might have this effect at a later time. There is no real reason to distinguish between the two situations since a fault does not change just because it has caused an error. We need only have one word: fault.

2.2 Avizienis

The description given here of the work of Avizienis is taken mainly from two papers[4,1]. "A *system* is an interacting set of subsystems embedded in an environment that provides inputs to the system and accepts output from the system. A subsystem itself is composed of subsystems, and so on, to a desired level of decomposition onto the smallest meaningful elements". Avizienis then goes on to define the *expected behaviour* of a system as "a time-dependent sequence of output states that agrees with the initial specifications from which the system has been derived".

We believe that there are two problems with this definition. For the first let us consider an interactive computer system. When we are typing at a terminal we usually see the characters we are typing being echoed on the screen. This typing of a character followed by an echo is two events, an input followed by an output. However, we often abstract

from sets of events like this and view them as single events. In this example, when we use a terminal we normally see the typing of a character followed by its echoing to the screen as a single input event. If the system did not echo the character we would look on it as failure of the system at an input event (at this level of abstraction), not as a failure at an output event. So in order to discuss failure and other reliability concepts a characterisation of behaviour in terms of outputs alone is not good enough. We need to consider inputs as well. Of course we need only consider outputs if the system is an "output only" system, for example a clock.

The second problem is that of state. It is of vital importance to differentiate between the state of a system and its output. We will illustrate this with an example; consider a system which has three lights, red, green and blue, and these flash in the order red, green, blue, green, red, green, blue, green, red, ... This system has three outputs (red, green and blue), but four states (which we can call red, blue, green-going-to-red and green-going-to-blue). When it is outputting green it can be in one of two distinct states. That is, the current or last output of a system is not sufficient to determine the state of the system nor to predict the expected future behaviour.

So, we would prefer to have the expected behaviour defined as a time-dependent sequence of events (and values) at the system interface that agrees with the initial specifications from which the system has been derived.

An *error* is defined to be an *undesired event* that occurs when the actual system behaviour

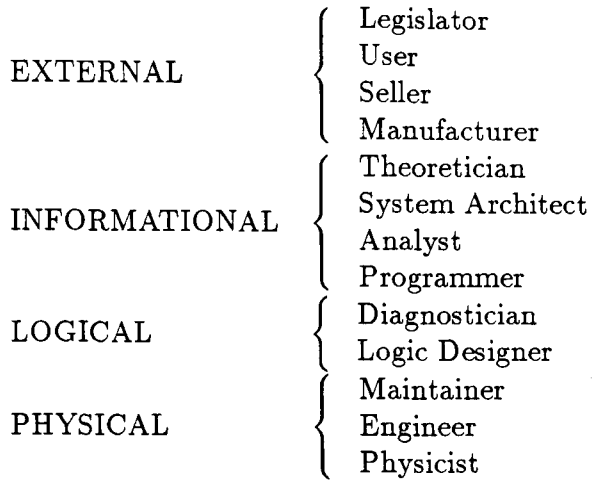


Figure 2.1: The Four Universe Model

deviates from the expected behaviour. This definition differs markedly from the one we will use. A *fault* is then defined as the cause of an error and a *failure* is the occurence of an error or errors that is specified to be a failure condition. An error is recognised as a failure by failure criterion

Based on these definitions Avizienis moves on to present a model representing the different views people have of the same system. He calls this the "Four Universe" model because it is based on the four universes of observation and interpretation: (1) physical; (2) logical; (3) informational; and (4) external (see figure 2.1).

This structure has some similarity to the ideas of interpretive interfaces[16] as used by Anderson and Lee. That is, we can view each layer (universe) as being interpreted by the layer below.

Avizienis argues that each one of these four universes contains its own consistent set of fundamental concepts, models and terminology. The design requirements, performance measures, forms of expected behaviour, test procedures and acceptance criteria can be completely stated in terms of the given universe. Avizienis then argues that we can use the terms that are already in accepted use for each of the different universes. The problem with this approach is that people do not work purely within a single universe, terms are needed that are consistent and applicable across all universes.

2.3 Anderson and Lee

The following description of the definitions used by Anderson and Lee is taken mainly from one paper[2] and a book[3]. A system is defined as "any identifiable mechanism which maintains a pattern of behaviour at an interface with its environment". A system consists of components (subsystems) which interact under the control of a design.

"The external behaviour of the system can be described in terms of a finite set of *external states*, together with a function defining transitions between states. The environment provides inputs as stimuli and perceives the system as passing through a sequence of external states at discrete instances of time". Again, as we did with Avizienis' defintions, we would question whether this is right. We need to distinguish between states and outputs. The environment cannot perceive the state of the system; it can only perceive

outputs and so we, part of the environment, can deduce the state.

The concept of *the design* is important in the definitions of Anderson and Lee. The design of a system is that part of the system which actually supports and controls the interaction of the components. The design needs to be part of the system because a design fault is a defect which is actually present in a system. Internal state transitions are a consequence of changes of state by the components; these changes are determined by interactions between the components. The pattern of these interactions is specified and controlled by the design of the system, which also determines the way in which interactions between the system and the environment impinge upon the components.

Anderson and Lee consider levels of abstraction within a system. That is they can regard a system in many different ways. For example, a computer system can be regarded in terms of circuit boards, chips and wires or as set of communicating processes.

A particularly important change in view can be identified at an interpretive interface where a component of a system is interpreted (i.e. executed) by, and thereby governs and directs the operation of, the rest of the system. When this is the case Anderson and Lee regard the interpreted component as the design of a system whose components are abstractions of the other components of the original system.

The paradigm here is, of course, the interface between hardware and software in a computing system. It will often be more useful to regard software as the design of an abstract

system rather than merely as a bit-pattern.

In order to define when a system fails, corresponding to Avizienis's failure criterion, Anderson and Lee have an *authoritative specification* which can be applied as a test in any situation to determine whether the behaviour of the system should or should not be deemed acceptable. For the purpose of definition the role of the specification is absolute.

They do not recognise that, since the only role of the authoritative specification is to identify acceptable/unacceptable behaviours, it does not have to be in a form which would allow the easy construction of a system. In fact the authoritative specification does not need to exist when construction of the system begins.

A *failure* is said to occur when the behaviour of the system first deviates from that required by the specification of the system.

An occurence of system failure must be due to the presence of defects within the system. Such deficiencies are referred to as faults when they are internal to a component or the design, or as errors when the state of the system is defective.

When a system fails its state will, at some previous point in time, have deviated from what is valid, that is the system must have entered an *erroneous state.* This will have occured at some time before (or possibly when) the system fails. This occurs either because a component fails or because there is a problem in the design of the system. An *error* is the

part of the state which would have to be changed to make the state valid. There could be a choice of parts which might be changed to make the state valid, which of these is to be called the error is a subjective judgement.

2.4 Laprie

Laprie introduces a 'new' word *dependability* which he defines as "the quality of the delivered service such that reliance can justifiably be placed on this service".

Laprie goes on to define a *failure* as a deviation of delivered service from that specified. A *fault* (eg programmer's mistake, short circuit on VLSI device) as the cause of a *latent error* (eg erroneous code, gate stuck at zero) which becomes an *effective error* (eg by execution of program module with appropriate input, by attempt to use stuck gate) when *activated* and may then result in a failure if steps are not taken by the system to render it latent once more.

An *error* is the manifestation *in the system* of a fault, and a *failure* is the manifestation *in the service* of an error.

We would argue that we do not need to have these concepts of latent and effective errors. Firstly, we can consider carefully what we mean by the state of a system, use that to define error and we make use of the concept of error propagation to show how an error can, over

time, affect larger and larger parts of a system and so lead to failure. This removes the need to distinguish between latent errors and faults. Secondly, there is a problem if we consider Laprie's examples for fault and latent error. He says that a short circuit on a VLSI device is a fault and a gate stuck at zero is a latent error. From these examples it seems strange to distinguish a fault from a latent error. The short circuit and the gate stuck at zero they may be the same behaviour seen from different viewpoints. Hence, we will make use of only two concepts, excluding failure: error, for something 'wrong' in the state of the system; and fault, for something 'wrong' in the structure of system. (In later papers than those considered here [5] Laprie does not have the the concepts of latent and effective errors.)

Laprie also develops a model of structure. This is based on two transverse relations *details* and *uses*. The *details* relation is one which addresses what makes up a system, details defines an order so we can talk about higher and lower in the details hierarchy. For example, consider a computer which consists of a CPU, semiconductor memory, an IO driver and a disk. The CPU and the memory details the 'processor', the IO driver and the disk details the 'mass storage', and the processor and the mass storage details the computer. The uses relation embodies the concept of *"uses the service of"*. A processor uses the service of a disk, a program uses the service of the operating system. The reason that the relations are transverse is that something higher in the details hierarchy cannot use something lower and something lower cannot use something higher. For example, a computer system does not use its processor, because the processor details the computer

system.

Laprie defines two further concepts, these are *layer* and *component.* "If at a 'level' all the elements can be ordered according to the relation *uses* then the elements are *all* layers otherwise they are *all* components."

This model has a number of problems. The first of these problems is that Laprie does not properly define what is meant by a *level* in the tree formed by the details relation. We assume that a level is a set of elements with a common predecessor. It is not clear whether two layers in different levels can be related though one could assume this given the sentence "a component in a given layer may use the services of several components of lower layers and the service it delivers may be used by several components of upper layers." If one can relate the elements in different layers no consistency rules are given. Laprie argues that an interpreter can be viewed as one layer and the program it interprets as a higher layer. It is hard to see which way round the *uses* relationship should go. Does a program use an interpreter to be executed or does an interpreter use a program in order to know what to do.

If we try and relate the ideas of the previous paragraph to the models developed in this thesis we encounter a further problem, as the program is part of the state of its interpreter and we find it difficult to consider it as a separate component.

2.5 Kopetz

Kopetz does not consider systems in general but identifies a particular type of component of a system which he wishes to consider. This component is a stored program computer, which we will abbreviate to SPC. Every SPC is embedded in a process of its environment which provides input data to and accepts output data from this SPC. It is assumed that there exists an authoritative specification for the operation of an SPE and some expectations about the behaviour of its environment. This second assumption allows Kopetz to have a concept of an input fault, that is when the environment provides an input which is outside those defined by the expectations we have of its behaviour.

An SPC carries out computation and consists of a machine and two data structures, which are called the i-state and the h-state. The machine contains a set of processors which execute the stored program and a read/write memory which can store programs and data. The i-state specifies the contents of the memory which are not changed during computation; that is, the i-state is static. The h-state comprises all information which has been accumulated in the history of the SPE. The h-state is dynamic and can be changed by computation. The computation consists of a sequence of activities which take input and the current h-state and from this generate an output and a new h-state. Activities have duration whereas events, such as the start or finish of an activity, have no duration. Figure 2.2, from [17] shows this model of a computation.

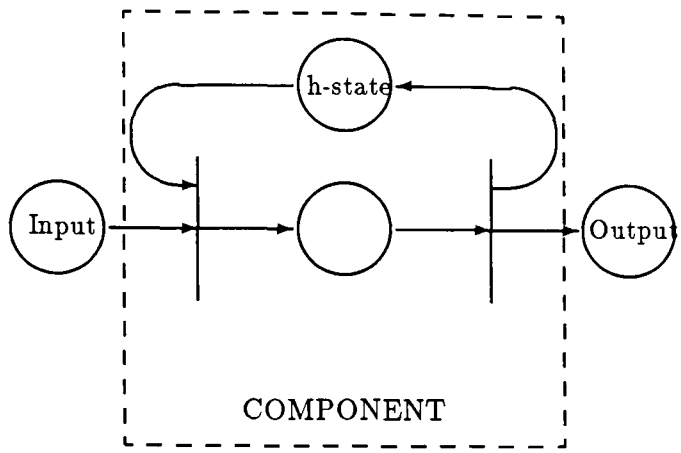


Figure 2.2: A Model of Computation

A failure is an event, i.e. a happening at a given point in real-time, while a fault is a state, i.e. an attribute value of an object which is invalid for a certain time interval. Since a state change is an event, two events can be associated with every fault: the fault starting event and the fault finishing event. A failure is always a fault starting event. The fault terminating event can be the completion of a repair action or some other event which causes the disappearance of the fault.

If, during the duration of a fault, an activity interacts with the object which is at fault a consequent failure of this activity may occur. That is from the point of failure onwards the activity will be at fault. This new fault will remain in the system until its terminating event has occured.

Kopetz applies the word fault to invalid state. This distinguishes him from the majority of others who use the term error. However, because Kopetz views the program as part of the state of his SPC the word fault can be applied to something being wrong with the program (i-state). Though we use both fault and error in our definitions we recognise that they are two names for the same basic concept, namely some condition of the system not being as it should be.

2.6 Robinson

In most papers which deal with defining the concept of failure there is an assumption that we have some authoritative specification with which we can compare system behaviour. A S Robinson looks at this assumption and considers what constitutes an "authoritative system reference (ASR)", as he calls it. He also considers the problems of the many levels in a computer system, the physical level, the logic level etc (after Avizienis) and asserts that when a system is being constructed there will be many ASR's, probably one for each level, which together build up into an overall ASR. For example, a computer system has a specification for its hardware and (possibly) another for its software. This means that when we are discussing errors and failures in a system we need to know which level and thus which ASR we are dealing with.

"The term *error* is used to indicate that an action or a value is not as it should be. Thus,

an error is made (action); a value is in error (state)". The explanation given for this use is that Robinson is trying to produce definitions that are 'consistent' with "general engineering usage". The use of error to refer to two distinct concepts, namely an action and a condition, overloads a word which is misused in too many other ways already.

A failure is defined as "an event corresponding to the first occurrence of a generated output error" and a fault is defined as "a defect in a physical system that has the potential for generating errors in operational use". Both these definitions suffer from a lack of rigour. Is the output error an event or a state? If it is a state then our arguments earlier concering the need to distinguish between the output and the state apply. If it is an event then a failure could have been defined as: the first occurrence of a perceivable error. What is meant by a *physical* system? Does Robinson intend that a program cannot contain a fault?

2.7 Cristian

Our discussion is based on an IBM Technical Report[10]. Cristian, because of his view that there is a substantial difference between hardware and software faults, does not attempt to give definitions that cover both the physical nature of hardware faults and the logical nature of software faults. He assumes ideal (fault-free) hardware and focuses only on the software issues. He considers only sequential programs and gives mathematically

rigorous definitions using a relational approach to the semantics of programs[6].

He defines the *standard domain* for a program as the set of initial states for which the program terminates and for which the initial and final states satisfy the program's specification. The *failure domain* is the set of initial states which are not in the standard domain. (Cristian also consider exception handling and this adds additional domains.)

A *design* is a pair consisting of a specification and a program.

An invocation of a program in its failure domain is a *failure occurence.* This definition does not imply that failure occurences are actually recognised. If the failure domain of a program is non-empty then its design is said to contain a *design fault.* This may be a specification fault, if no program response is specified for a possible input, or a coding fault, if the result actually produced by the program contradicts the specification.

Consider the situation where a coding fault exists in a program. It is then possible that an invocation of the program from some initial state terminates in some final state that is different to the final state prescribed by the specification. Such a final state is an *erroneous state.* In a erroneous state there exist variables that have values which are different from the values they should have, as defined by the specification. A variable value that is different from the value that a specification prescribes is an *error.* In this case, one cannot properly define errors in the case of specification faults since the final state is undefined, hence the proper values for the variables are undefined.

The definitions of reliability concepts by Cristian are rigorous and precise and where possible we have attempted to remain consistent with his definitions. This has not been entirely possible due to the greater complexity of the object we have been trying to model, namely a system. However, his concept of state and error has influenced our approach.

2.8 BS5760/BS4778

BS5760 "Reliability of Constructed or Manufactured Products, Systems, Equipments and Components" describes the concepts associated with software reliability using the diagram in figure 2.3. BS5760 does not explain the 'revealing mechanism'. How does the environment interact with the system in such a way that faults within the system manifest themselves? An error is defined as "a human action that produces an unintended result", a fault as "an incorrect portion of software which has the potential of causing failure" and a software failure as "a failure caused by a software fault". A failure is defined as "a deviation of system behaviour from that required".

BS5760 also goes on to say: "It should be remembered that software itself does not fail. Faults already present in software lead to failure of the system under certain conditions." The reason for this is to remain consistent with the ISO (BS4778) definition for a failure: "the termination of the ability of a functional unit to perform its required function". That is, the unit could perform its function to specification, all valid inputs would produce the

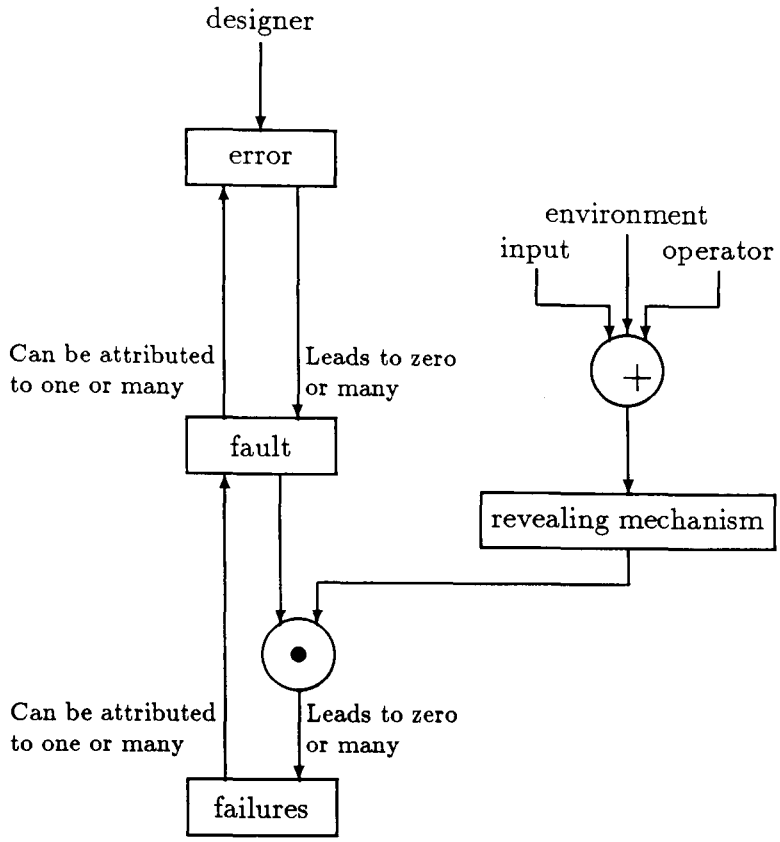


Figure 2.3: Error, Fault, Failure relationship

correct output. Then at some point in time some failure event occurs, internal to the unit, and from that point onwards the unit would be unable to act according to its specification. An example of a failure event would be a change in the structure of the unit due to physical aging and wearout. Such a definition is not applicable to software. There is no *internal* mechanism which can turn software which meets its specification into software which doesn't.

Chapter 3

System Structure

In this chapter we consider the structure of objects. We clarify an aspect of the concept of abstraction, that of collapsing or aggregation, by considering two views of structure. The approach we take is to first consider one of these views and a corresponding formalization, namely directed graphs; the other view is then examined and we show that it is simply a special case of the former. It is a special case in that rather than defining a general directed graph it defines a tree.

This leads us to the concept of "collapsing", a technique of abstraction which groups objects together and "collapses" them to a single object. We formalize this by developing a model of structure called "structured graphs".

Finally we use structured graphs to produce a mathematical model of atomic actions, correcting a previous model[6].

The following definition is by Parnas[22].

Definition 3.1 Structure *refers to a partial description of an object showing it as a collection of parts and showing some relations between the parts.*

The obvious way to formalize this is to use a pair (X, U) where X is a set representing the parts and $U \subseteq X \times X$ is a set representing the relations between the parts. The pair (X, U) is a directed graph.

3.1 Directed Graphs

The notation we use in our presentation of directed graphs is drawn from [9], as are most of our basic definitions. Also, since we deal only with directed graphs, we shall simply use the word graph when referring to directed graphs.

Definition 3.2 *A* graph $G = (X, U)$ *is an ordered pair, where* X *is a finite non-empty set of* nodes *and* U *is a finite set of* arcs, *with* $U \subseteq X \times X$.

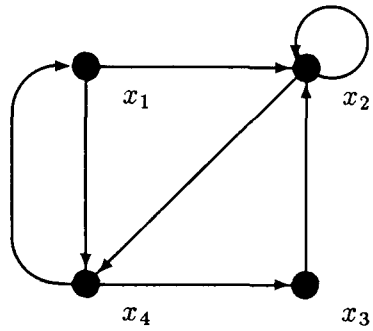


Figure 3.1: Graph of example 3.1

A graph can be depicted by a diagram in which nodes are represented by points in the plane and each arc $(x, x') \in U$ is represented by an arrow drawn from the point representing x to the point representing x'.

Example 3.1 Figure 3.1 represents the graph $G = (X, U)$ where $X = \{x_1, x_2, x_3, x_4\}$ and $U = \{(x_1, x_2), (x_1, x_4), (x_2, x_2), (x_2, x_4), (x_3, x_2), (x_4, x_1), (x_4, x_3)\}$.

Definition 3.3 *In a graph $G = (X, U)$ a node x is called a* successor *of a node x' if $(x', x) \in U$. Similarly, a node x is called a* predecessor *of x' if $(x, x') \in U$.*

The set of all successors of x is denoted by $\Gamma^+(x)$, the set of all predecessors by $\Gamma^-(x)$. We denote $\Gamma^+(x) \cup \Gamma^-(x)$ by $\Gamma(x)$, the set of all neighbours *of x.*

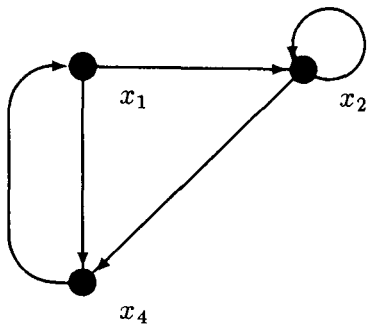


Figure 3.2: Subgraph of example 3.3

Example 3.2 In the graph of example 3.1 the node x_2 has successors x_2 and x_4, and predecessors x_1, x_2 and x_3.

Definition 3.4 *Let $G = (X, U)$ be any graph then, given a set Y such that $Y \subseteq X$, we define the* subgraph H *of G generated by Y as $H = (Y, U_Y)$ where $U_Y = U \cap (Y \times Y)$.*

Example 3.3 Figure 3.2 shows the subgraph of Figure 3.1 generated by $\{x_1, x_2, x_4\}$.

Definition 3.5 *A* path m *is defined as a finite set of arcs of the form:*

$$m = (x_0, x_1), (x_1, x_2), \ldots, (x_{r-1}, x_r)$$

The number of arcs in the sequence is called the order *of the path. The node designated by x_0 is called the* initial endpoint *of the path and the node designated by x_r is the* final

or terminal endpoint *of the path.*

A path whose endpoints are distinct is said to be open*; whereas one whose endpoints coincide is called a* closed path *or* cycle.

We observe that a path is completely determined by the sequence of nodes which it visits. We shall often find it convenient to specify a path by listing this node sequence rather than the more strictly correct arc sequence.

Example 3.4 In the graph of figure 3.1 the arc sequence:

$$(x_1, x_4), (x_4, x_3), (x_3, x_2), (x_2, x_4), (x_4, x_1), (x_1, x_2)$$

is a path of order 6. The path can also be described by the node sequence:

$$x_1, x_4, x_3, x_2, x_4, x_1, x_2$$

. The node sequence x_2, x_4, x_1, x_2 is a cycle of order 3.

Definition 3.6 *Let $G = (X, U)$ be any graph, and let x be one of its nodes. Then any node x' (not necessarily distinct from x) for which there exists a path from x to x' is called a* descendant *of x; while any node x' (not necessarily distinct from x) for which there exists a path from x' to x is called an* ascendant *of x.*

It will be observed that a node x' can be both a descendant and an ascendant of x; this occurs whenever there exists a cycle passing through both x and x'.

We shall denote the set of descendants of a node x by $\hat{\Gamma}^+(x)$ and the set of ascendants by $\hat{\Gamma}^-(x)$.

A node x' is said to be *accessible* from a node x if x' is a descendant of x or $x' = x$; similarly, x' is said to be *converse-accessible* from x if x' is an ascendant of x or $x' = x$. The set of nodes which are accessible and converse-accessible from x will be denoted by $\overset{*}{\Gamma}^+(x)$ and $\overset{*}{\Gamma}^-(x)$ respectively.

Clearly

$$\overset{*}{\Gamma}^+(x) = \{x\} \cup \hat{\Gamma}^+(x)$$

$$\overset{*}{\Gamma}^-(x) = \{x\} \cup \hat{\Gamma}^-(x)$$

We will also denote $\hat{\Gamma}^+(x) \cup \hat{\Gamma}^-(x)$ by $\hat{\Gamma}(x)$ and $\overset{*}{\Gamma}^+(x) \cup \overset{*}{\Gamma}^-(x)$ by $\overset{*}{\Gamma}(x)$.

When needing to refer to more than one graph we will use the graph as a subscript. For example $\overset{*}{\Gamma}^+_T(x)$.

Example 3.5 On the graph $G = (X, U)$ of figure 3.1, $\hat{\Gamma}^+(x) = X$ and $\hat{\Gamma}^-(x) = X$, for all $x \in X$, since all nodes $x \in X$ are on paths to all other nodes.

Graphs which contain no cycles are often useful; such graphs are given a name.

Definition 3.7 *An* acyclic graph *is a graph which does not contain any cycles, that is, one for which for all nodes x, $x \notin \hat{\Gamma}^+(x)$.*

When we examine an object, and consider its structure, we usually refer to the parts of the objects as components. Furthermore, we can often identify levels of components or, at least, we often refer to components being below (or above) other components. That is, the object is structured as a hierarchy of components. When an object is structured as a hierarchy the relationships between the components define an order, below or above for example. It can be shown than any order defines an acyclic graph, though we do not do so here, and so we reach the following definition.

Definition 3.8 *An object is* hierarchically structured *when the graph representing the structure of that object is acyclic.*

Another concept we will make use of is that of a "cut". Although this concept will only be applied to trees (to be defined), we will define it here in its full generality. A cut breaks a graph into three parts, those elements which are in the cut, those which are 'before' the cut and those which are 'after' the cut.

Definition 3.9 *Let $G = (X, U)$ be a graph and $C \subseteq X$; let*

$$C^+ = \bigcup_{x \in C} \hat{\Gamma}^+(x)$$
$$C^- = \bigcup_{x \in C} \hat{\Gamma}^-(x)$$

C is a cut *through G if and only if*

$$C \neq \emptyset$$

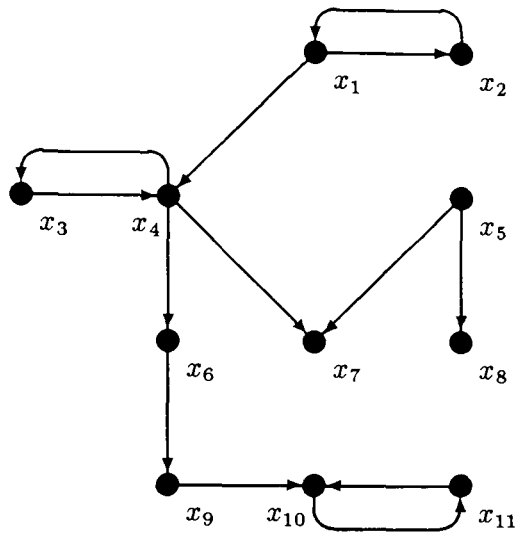


Figure 3.3: Graph of example 3.6

$$C \cap C^+ = C \cap C^- = C^- \cap C^+ = \emptyset$$

$$C \cup C^+ \cup C^- = X$$

Example 3.6 $\{x_6, x_7, x_8\}$ is a cut through the graph of figure 3.3.

There is a special case of an acyclic (directed) graph which we will make use of. We will introduce it by example.

Let us consider a book; we can say a book 'contains' Chapter 1, Chapter 2 etc. and Chapter 1 'contains' Section 1, 2 etc. If we draw the graph of 'contains', Figure 3.4, we

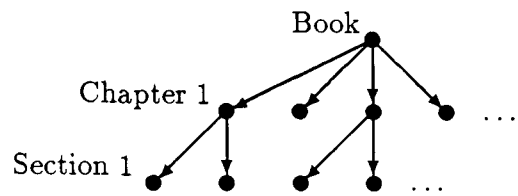


Figure 3.4: A Book

find it looks like an upside down tree.

Definition 3.10 *A* tree *is an acyclic graph in which just one node has no predecessors (this node is known as the* root*), and every other node has exactly one predecessor. The set of nodes which have no successors are called the* leaves *of the tree.*

Definition 3.11 *Let $G = (X, U)$ be tree, then a* sub-tree *of G is a sub-graph of G which is also a tree.*

Horning and Randell[15] consider structure to be "tree" based, each component of an object consisting of further components, this subdivision continuing until we wish to stop. The undivided components are normally referred to as atomic (not to be confused with atomic actions, which are discussed later in this chapter). It is immediately obvious that this is a special case of definition 3.1 with the relation 'is a component of'. The corresponding graph is, of course, a tree.

Let us again consider the example of a book, which consists of chapters, where each chapter consists of sections and each section consists of paragraphs; there is obviously a tree structure for the book described by 'consists of'. There is also a relation 'occurs before' between the components of the book. We can say that chapter 4 occurs before section 5.1 but we cannot say section 3.2 occurs before chapter 3, since these two are not 'comparable'. The condition for comparability between two components x and y is $x \notin \hat{\Gamma}(y)$

We are moving to define two relations over a set of elements, one relation is of a 'consists of' or 'contains' type and defines a tree, and the other relation can be considered in some sense 'orthogonal' to the first, for example 'uses' or 'occurs before'.

If we consider as another example a computer program and assume there is no nesting of procedures, we can describe a 'uses' relation between the procedures. We can group together those procedures which are related by some criterion. We can repeat this process, grouping together related groups, until we have only a single object to consider; we are establishing a tree structure for the program. We will find that at any stage in this grouping the relation 'uses' can easily be kept meaningful by using the rule: if a component of x 'uses' a component of y then x 'uses' y, provided that x and y are distinct.

The reason for the final proviso is that when viewing an object from "outside" we should not be interested in its internal structure and so x 'uses' x will be irrelevant. The extended 'uses' relation obeys the condition for comparability given above and in the next section

we consider a formalization of objects structured by two relations one of which defines a tree.

3.2 Structured Graphs

We wish to be able to group together parts of our graph to hide "details". The approach we will take to this is to pick out the subgraph representing the part we wish to group together and replace that subgraph with a single new node, all arcs leading into and out of the subgraph are replaced by arcs ending and starting with the new node. We can then choose another part we wish to "collapse" and repeat the process on that part. We demand that the "collapsing" must occur one subgraph at a time so that overlapping of our newly introduced nodes is precluded. This means that we get a well nested structure and so the collapsings can be represented by a tree.

Example 3.7 Figure 3.6 shows the graphs obtained by the "collapsings" of the graph on the left in figure 3.5, according to the tree on the right in figure 3.5.

The above leads us to the following definition.

Definition 3.12 *A* structured graph $S = (T, G)$ *is an ordered pair where* $T = (E, U)$ *is a tree, called the* structure tree, *and* $G = (B, R)$ *is a graph, called the* basic graph, *where*

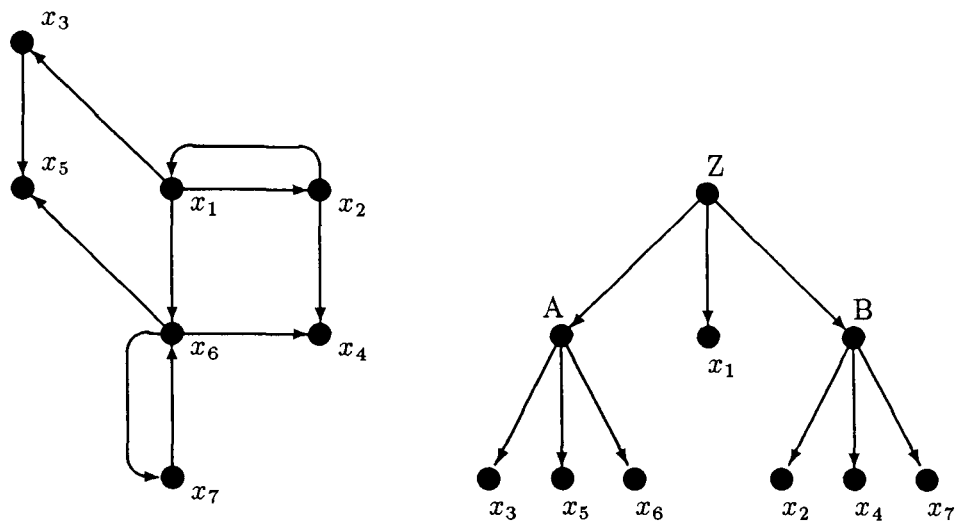


Figure 3.5: A Structured Graph

$B \subseteq E$ is the set of leaves of T.

Definition 3.13 *Let $S = (T, G)$ be a structured graph, with $T = (E, U)$ and $G = (B, R)$ and let $x \in E$, then we define $\hat{x} = \overset{*}{\Gamma}{}^{+}_{T}(x) \cap B$. $\hat{x}$ is the set of leaves of the subtree of T rooted at x.*

Example 3.8 The leaves of the subtree rooted at A in figure 3.5.

$$\hat{A} = \{x_3, x_5, x_6\} \cap \{x_1, x_2, x_3, x_4, x_5, x_6, x_8\} = \{x_3, x_5, x_6\}$$

Definition 3.14 *Let $S = (T, G)$ be a structured graph, with $T = (E, U)$ and $G = (B, R)$,*

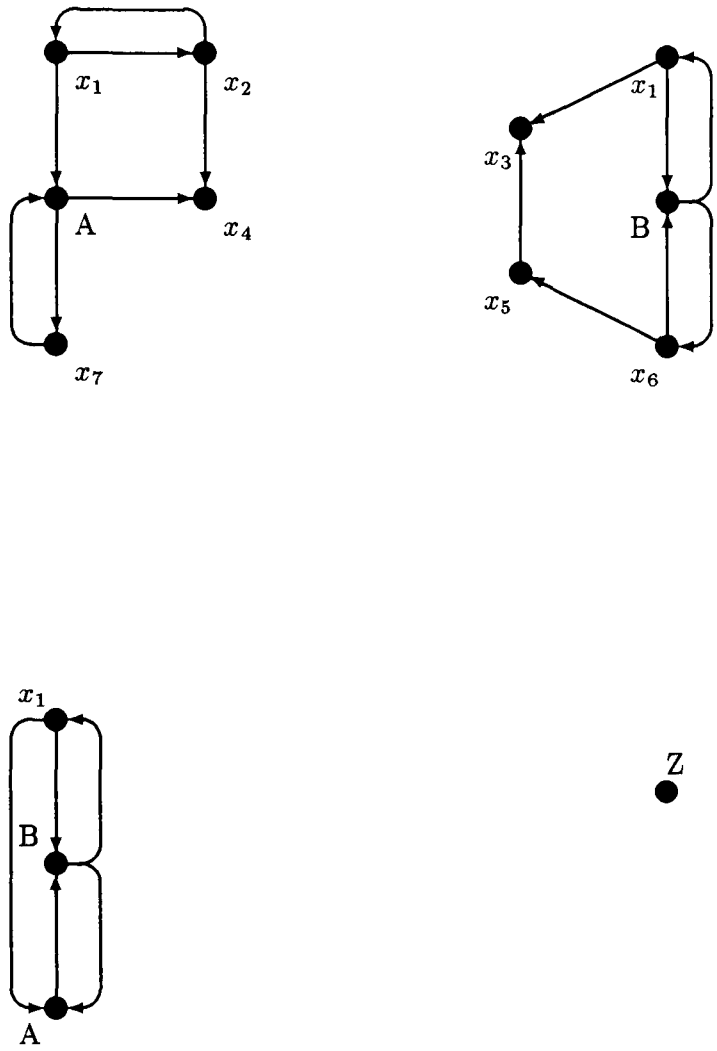


Figure 3.6: Collapsed Graphs

and let C be a cut through T, then we say that the graph G', defined by $G' = (C, P)$ where

$$P = \{(x,y) \in C \times C : x \neq y \text{ and } (\exists m \in \hat{x}, n \in \hat{y} : (m,n) \in R)\} \cup$$
$$(x,x) \in C \times C \cap R$$

is a level of detail *of S.*

Example 3.9 The graphs of figure 3.6 are levels of detail of the structured graph represented by figure 3.5.

Lemma 3.1 *The basic graph is a level of detail.*

Proof Let $S = (T, G)$ be a structured graph with $T = (E, U)$ is a tree and $G = (B, R)$. B is cut through T and

$\{(x,y) \in B \times B : x \neq y$ and $(\exists m \in \hat{x}, n \in \hat{y} : (m,n) \in R)\} \cup (x,x) \in C \times C \cap R$

$= \{(x,y) \in B \times B : x \neq y$ and $(x,y) \in R\} \cup (x,x) \in C \times C \cap R$, since $\hat{z} = \{z\}$ when $z \in B$

$= \{(x,y) \in B \times B : (x,y) \in R\}$

$= R$, since $R \subseteq B \times B$ □

Lemma 3.2 *Let x be the root of the structure tree. The graph $(\{x\}, \emptyset)$ is a level of detail.*

Proof Obvious □

Definition 3.15 *Let $S = (T, G)$ be a structured graph and let $G' = (B', R')$ be a level of detail of S. Also let $T = (E, U)$ and choose a non-leaf node $A \in E$ such that $\Gamma_T^+(A) \subseteq B'$. We define the* collapsing *of A in G' as the construction of the graph $G'[A]$ where*

$$
\begin{aligned}
G'[A] &= (B'[A], R'[A]) \\
B'[A] &= (B' \setminus \Gamma_T^+(A)) \cup \{A\} \\
R'[A] &= \{(x, y) \in B'[A] \times B'[A] : (x, y) \in R'\} \cup \\
&\quad \{(x, A) : x \in B'[A], (\exists y \in \Gamma_T^+(A) : (x, y) \in R')\} \cup \\
&\quad \{(A, y) : y \in B'[A], (\exists x \in \Gamma_T^+(A) : (x, y) \in R')\}
\end{aligned}
$$

Example 3.10 Consider the two graphs on the left in figure 3.6. The bottom graph can be created by the collapsing of B in the top graph.

Lemma 3.3 *Let S be a structured graph, $G' = (B', R')$ be a level of detail of S and A be a non-leaf node in the structure tree of S such that $\Gamma_T^+(A) \subseteq B'$. $G'[A]$ is a level of detail of S.*

Proof

Let $S = (T, G)$ with $T = (E, U)$ and $G = (B, R)$. Let $G' = (B', R')$. It can be seen that $B'[A]$ is a cut.

$R'[A]$

$=$ { By definition }

$\{(x,y) \in B'[A] \times B'[A] : (x,y) \in R'\} \cup$

$\{(x,A) : x \in B'[A], (\exists y \in \Gamma_T^+(A) : (x,y) \in R')\} \cup$

$\{(A,y) : y \in B'[A], (\exists x \in \Gamma_T^+(A) : (x,y) \in R')\}$

$=$ {G' is a level of detail, so $(x,y) \in R' = ((x,y) \in B' \times B') \wedge (\exists m \in \hat{x}, n \in \hat{y} : (m,n) \in R)$)}

$\{(x,y) \in B'[A] \setminus A \times B'[A] \setminus A : (\exists m \in \hat{x}, n \in \hat{y} : (m,n) \in R)\} \cup$

$\{(x,A) : (x \in B'[A] \setminus A) \wedge (\exists m \in \hat{x}, n \in \hat{A} : (m,n) \in R)\} \cup$

$\{(A,y) : (y \in B'[A] \setminus A) \wedge (\exists m \in \hat{A}, n \in \hat{y} : (x,y) \in R)\}$

$=$

$\{(x,y) \in B'[A] \times B'[A] : (\exists m \in \hat{x}, n \in \hat{y} : (m,n) \in R)\}$ □

Lemma 3.4 *Let $S = (T,G)$ with $T = (E,U)$ and $G = (B,R)$. Let $G' = (B',R')$ and $G'' = (B'',R'')$ be levels of detail of S with $\Gamma_T^+(A) \subseteq B'$ and $\Gamma_T^+(A) \subseteq B''$. If $G'[A] = G''[A]$ then $G' = G''$.*

Proof Obvious, from the definition of collapsing □

Definition 3.16 *Let $S = (T,G)$ be a structured graph and let $G'' = (B'',R'')$ be a level of detail of S. Also let $A \in B''$, with A a non-leaf node. We define the* opening *of A in G'' as any level of detail $[A]G''$ with the property $([A]G'')[A] = G''$.*

Lemma 3.5 *Let $S = (T,G)$ be a structured graph and let $G'' = (B'',R'')$ be a level of*

detail of S. Also let $A \in B''$, with A a non-leaf node. $[A]G''$ exists and is unique.

Proof Let us consider the level of detail G' defined by $G'[A] = G''$. Then $([A]G'')[A] = G'$ and so an $[A]G''$ with the required property exists.

Let $[A]G' = G_1$ and $[A]G' = G_2$ be different. From the definition $G_1[A] = G' = G_2[A]$. By lemma 3.4 $G_1 = G_2$ and so $[A]G'$ is unique □

Lemma 3.6 *Let $S = (T, G)$, $T = (E, U)$, be a structured graph and let $G' = (B', R')$ be such that $B' \subset E$. G' is a level of detail of S if and only if there exists $A_1, \ldots, A_n \in E$ such that $G[A_1] \ldots [A_n] = G'$. That is, all levels of detail can be constructed by repeated collapsings and all graphs constructed by repeated collapsings are levels of detail.*

Proof The theorem is obviously true for the basic graph, for which $n = 0$.

That all repeated collapsings of the basic graph are levels of detail is proved by induction over the number of collapsings, with lemma 3.1 as the basis and lemma 3.3 as the inductive step.

The converse is proved as follows: take G', either $G' = G$ or there exists some node A_n of G' which is not a node of G, and we can then open A_n to produce a new level of detail $[A_n]G'$. This process can be repeated, but must eventually terminate since T is a finite tree. We will then have $G = [A_1] \ldots [A_n]G'$ and so $G[A_1] \ldots [A_n] = G'$ □

3.3 Atomic Actions

Intuitively, an atomic action is a group of events that enjoys the status of a simple "primitive" with regard to its environment. At the same time it may, however, possess a "complicated" internal structure (this is the same use as is found in [20] and [6]). By primitive we mean that no event outside the action *needs* to occur during the action. If such events exist we say that they *interfere* with that action.

Atomic actions are important in the study of the dynamic behaviour of systems. They have been used, for example, in the analysis of orphans in a distributed system [24] and, widely, in database systems, where they are more commonly called "transactions" [13].

We develop two dynamic atomicity criteria; they are dynamic because we take individual computations as our basic objects. We will use occurrence graphs, a particular type of directed graph, as our formal model of computations. An occurrence graph serves to model a single computation as a set of interdependent events, the dependencies arising from the semantics of a particular computation. We represent the grouping of events by making use of structured occurrence graphs, an application of structured graphs.

The work presented here derives directly from the application of occurrence graphs to the formalization of the concept of atomic actions in [6]. One of our two criteria is in fact identical to one given in [6] and the other is closely related. Other formalisms could

be used to investigate the concept of atomicity. Structured occurence graphs are used because they are an example of structured graphs and because they relate easily to the work of Best on which our work is based.

We will first define structured occurrence graphs and then use them to formally define our two atomicity criteria. The first is a global atomicity criterion, related to "serialisability". The events in the computation have some ordering on their occurence. The serialisability criterion states when the computation can be realised by some total order of the events occurence, that is the events can occur one at a time in a particular sequence. This total order needs to be 'compatible' with the ordering. That is the computation carried out with no events occuring concurrently will 'produce the same result' as the computation restricted so that events can occur concurrently as defined by their ordering. Our global criterion is identical to the global criterion found in [6].

The second of our criteria is a local atomicity criterion, "interference freeness". Informally, no activity can exist outside of the atomic action which has to occur during it, i.e. interfere with it. Two lemmas are established, one connecting our local and global criteria and the other relating to the internal structure of atomic actions.

Our local criterion is a modification of the local criterion from [6]. We will therefore state the local criterion from [6] and discuss why a new one is required. A theorem is proved connecting these two local criteria.

Definition 3.17 *Let $S = (T, G)$ be a structured graph where $T = (E, U)$ and $G = (B, R)$. If elements of B represent events, the elements of R represent precedence relations between those events, and G is acyclic, then we say that S is a* structured occurrence graph.

We are dealing here with the occurence of events. So, our precedence relations represent the ordering of events in time.

Nodes of the structure tree represent events. Non-leaf nodes of the structure tree will be referred to as *actions*, they represent groups of events. Leaf nodes of the structure tree will be referred to as *basic events.*

Definition 3.18 *Let $S = (T, G)$ be a structured occurrence graph and let $G' = (B', R')$ be any level of detail of S with $x, y \in B'$.*

We say x immediately precedes *y, written $x \prec\cdot y$, if and only if $(x, y) \in R'$,*

x precedes *y, $x \prec y$, if and only if there exists a path in G' starting with x and finishing with y, and*

x concurrent with *y, $x \parallel y$, if and only if $x \not\prec y$ and $y \not\prec x$.*

Now, consider the structured occurrence graph of figure 3.7 and then look at the level of detail, which is shown in figure 3.8, defined by the cut $\{e_2, A\}$. We have, by the

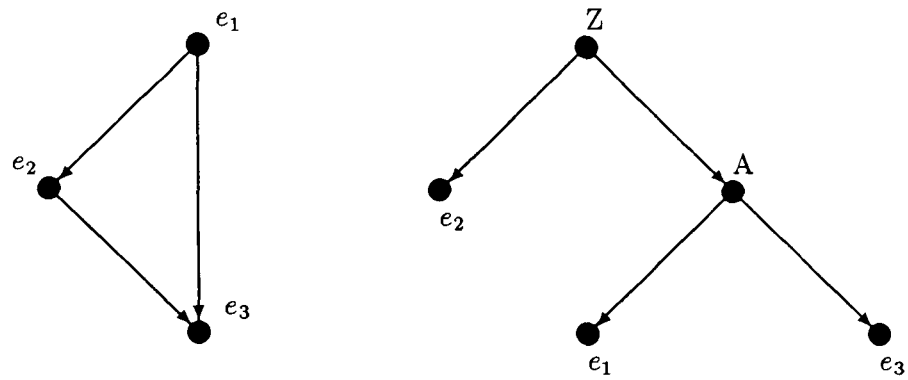


Figure 3.7: A Structured Occurence Graph

collapsing to create that level of detail, introduced a cycle. This cycle exists because e_2 occurs "during" A. That is, e_2 occurs after one event within A, e_1, but before another, e_3. So there can exist no total order for the events which can include both e_2 and A.

Definition 3.19 *Let S be a structured occurrence graph. We say S* satisfys atomicity *if and only if all levels of detail of S are acyclic.*

In a computation that does not satisfy atomicity we will find that some events are interfered with and some events interfere, others may do both. For example, from figure 3.7 and figure 3.8, we see that A is interfered with by e_2 and e_2 interferes with A. We will say that an event A is atomic if and only if it is not interfered with; that is, there does

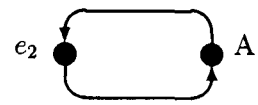


Figure 3.8: Level of Detail

not exist an event e, outside of A, that occurs after one part of A and before another. An elementary observation is that the elements of the basic graph G must be atomic, since there are no cycles in the basic graph of a structured occurence graph.

Definition 3.20 *Let $S = (T, G)$ be a structured occurrence graph where $T = (E, U)$, $G = (B, R)$ and let $L = (B', R')$ be a level of detail of S with $x, y \in B'$. Then y* interferes with x at L *if and only if $x \prec y \prec x$ and there exists an $x_1, x_2 \in \hat{x}$ such that $x_1 \prec y \prec x_2$ and $x_2 \not\prec x_1$ at the level of detail defined by the cut $(B' \setminus \{x\}) \cup \hat{x}$.*

This definition formalises our concept of interference. It states that an event y interferes with another event x if and only if there exists two basic events within x such that y has to occur before one and after the other. Using this definition we can state our local atomicity criterion.

Definition 3.21 *Let $S = (T, G)$ be a structured occurrence graph and D be the set of all levels of detail of S which include an event e. We say e is* atomic *if and only if for all levels of detail (B', R'), such that $e \in B'$, we have that for all $e' \in B'$, e' does not interfere with e at the level of detail (B', R').*

That is, an event is atomic if there is no event which interferes with it at any level of detail.

Lemma 3.7 *Let $S = (T, G)$ and $S' = (T', G)$ be structured occurrence graphs with $T = (E, U)$, $T' = (E', U')$, $G = (B, R)$ and $x \in E \cap E'$ such that $\hat{x}_T = \hat{x}_{T'}$, and the sub-trees generated by $(\bigcup x \in C : \overset{*}{\Gamma}^{-}_{T}(x))$ and $(\bigcup x \in C : \overset{*}{\Gamma}^{-}_{T'}(x))$, where C is the cut $(B \setminus \hat{x}) \cup \{x\}$, are equal. Then x is atomic in S if and only if x is atomic in S'.*

Proof Since the basic graphs of S and S' are equal and the structure trees excluding the sub-trees rooted at x are equal, any level of detail of S which contains x is a level of detail of S', and vice-versa.

We also have $\hat{x}_T = \hat{x}_{T'}$ so for any cut B' containing x, which will exist in both structure trees, the level of detail defined by $(B' \setminus \{x\}) \cup \hat{x}$ is the same in both S and S'.

Hence, from definitions 3.20 and 3.21, x is atomic in S if and only if x is atomic in S' □

The previous lemma shows that whether an event is atomic or not does not depend on

its internal structure, but only on the basic events that make it up and on the structure of the events outside it.

Example 3.11 Looking at the structured occurrence graphs of figures 3.9 and 3.10, we see that Z is atomic in both cases.

Our local and global atomicity criteria are related in a very obvious way as we will show in corollary 3.10.

Interference of x with y can also be characterized by the fact that, at some level L, x and y are on a cycle which is eliminated when y is opened, i.e. at $[y]L$.

Lemma 3.8 *Let $S = (T, G)$ be a structured occurrence graph and let $L = (B, R)$ be a level of detail of S with $x, y \in B$, and x and y on a cycle. Consider $[x]L$, if there does not exist an $x' \in \Gamma_T^+(x)$ such that x' and y are on a cycle then y interferes with x*

Proof

Cycle disappears

$\Rightarrow$

$\exists x', x'' \in \Gamma_T^+(x) : x' \prec y \prec x''$ and $x'' \not\prec x'$

iff { Opening up x' and x'' }

$\exists x_1 \in \hat{x}', x_2 \in \hat{x}'', x', x'' \in \Gamma_T^+(x) : x_1 \prec y \prec x_2$ and $x_2 \not\prec x_1$

iff { Eliminating x' and x'' }

$\exists x_1, x_2 \in \hat{x} : x_1 \prec y \prec x_2$ and $x_2 \not\prec x_1$

iff { By definition 3.20 }

y interferes with x □

Example 3.12 For the structured occurrence graph of figure 3.7, at the level of detail defined by $\{e_2, A\}$, e_2 and A are on a cycle which is eliminated when A is opened.

Lemma 3.9 *Let $S = (T, G)$ be a structured occurrence graph with $T = (E, U)$ and let $L = (B, R)$ be a level of detail of S:*

1. *If there is no level of detail in which $A \prec A$ then A is atomic.*

2. *If $e \prec e$ for some $e \in B$ then there exists an $e' \in B$ such that $e \prec e' \prec e$ at L and e' is not atomic. That is, if there is a cycle at L then some event in that cycle is not atomic.*

Proof

1. For something to interfere with A it must be on a cycle with it, by definition 3.20. So if A is not on a cycle then A must be atomic

2. Since the basic graph is acyclic, if there is a cycle there must exist some sequence of openings that make it disappear. Therefore, by lemma 3.8, there must be a

non-atomic event on that cycle.

□

Corollary 3.10 *A structured occurrence graph satisfies atomicity if and only if all of its events are atomic.*

Proof If the structured occurence graph satisfys atomicity then all levels of detail are acyclic and so at no level is any event involved in a cycle; therefore, by lemma 3.9(i), all events must occur atomically.

If all the events occur atomically then by lemma 3.9(ii), no event can be involved in a cycle at any level of detail, since if it was that would imply the existence of a non-atomic event; hence, all levels of detail must be acyclic and so the graph satisfys atomicity. □

We will now consider Best's local criterion from [6], which our local criterion was based upon, and explain why we felt a new criterion was necessary. Best gives the following, which has been altered to match our notation:

In a given structured occurrence graph (T, G),

1. Basic events occur atomically,
2. An action A occurs atomically if and only if

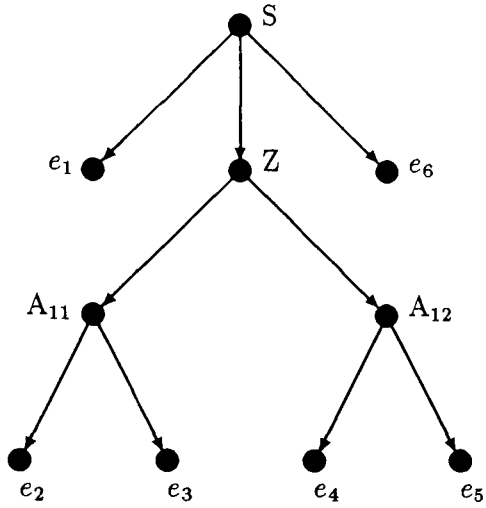
(a) $\forall a \in \Gamma_T^+(A)$, a occurs atomically, and

(b) for all levels L, whenever $e \prec A \prec e$ at L then $\exists a \in \Gamma_T^+(A)$ such that $e \prec a \prec e$ at $[A]L$.

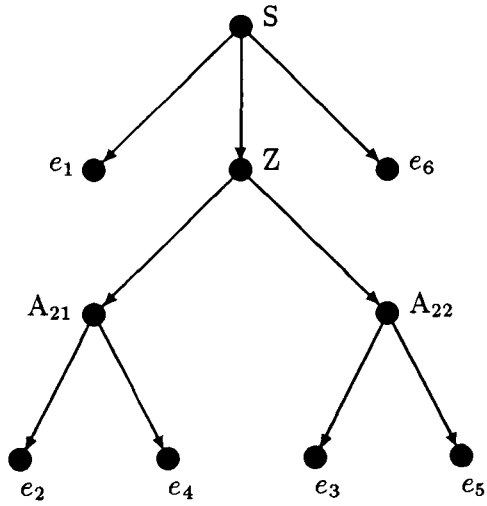
To illustrate what we consider to be wrong with this definition let us consider the structured occurrence graphs of figure 3.9 and figure 3.10. Consider all the levels of detail which contain Z. There is in fact only one and it is illustrated in figure 3.11.

Now we ask, is Z atomic? By our local criterion it is for both structured occurence graphs, since it never appears on a cycle. However, by Best's definition Z is not atomic in the structured occurrence graph defined by figure 3.9(b). This is because the events A_{21} and A_{22} are not atomic. We claim that since there is nothing that can interfere with Z, see figure 3.11, then this is not a sensible conclusion to reach.

A discussion of the extent to which any formal theory based on structured occurrence graphs can capture our intuitive notions of atomic actions can be found in [6] and [7].



(a)



(b)

Figure 3.9: Structure Trees

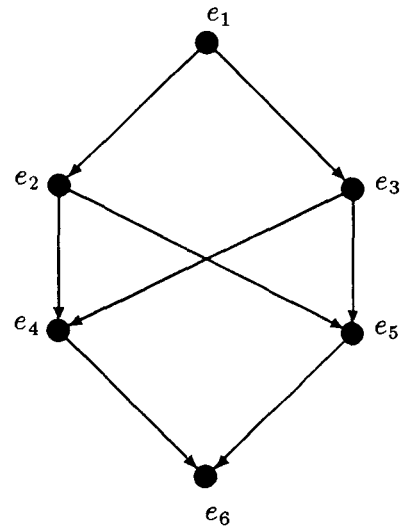


Figure 3.10: Basic Graph

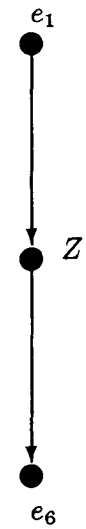


Figure 3.11: A Level of Detail

Chapter 4

System Behaviour

In this chapter we introduce a model of system behaviour based on the theory of traces. The theory of traces models behaviour in terms of the sequence of interactions taking place between the system and its environment. We consider the concept of abstraction when applied to the behaviour of a system and also the idea of programming a system.

A system is an object, either material or conceptual, having interacted, interacting or being able to interact with other objects, thus other systems. So, the basic property of a system is that it is (potentially) active.

We refer to the conglomeration of objects with which a system interacts as its environment; obviously this, viewed as a single object, is also a system.

Between the system and its environment there is a boundary; ie. a place, material or conceptual, at which the system "stops" and the environment "starts". Interactions between the system and its environment take place at this boundary and this is referred to as the system interface. We take an interaction to be the passage of some object through the interface.

What we have done is to split the "universe" into two duals, the system and its environment, which interact at a single interface. The reason we refer to them as duals is that if S is a system and E is the environment of S, then S is the environment of E.

When we consider the interface of a particular system we see that it consists of many places at which interactions can occur. We call these places ports . To simplify, we stipulate that objects may only pass through a particular port in one direction, either into or out of the system. Those ports through which objects pass into the system are called inports, those through which objects pass out are called outports. Objects which pass through a particular port will all belong to some related set of objects, this set is known as the sort of the port; c.f. the typing of variables in a programming language.

4.1 Traces

Definition 4.1 *An* interface N *is a triple* $(\mathbf{i}N, \mathbf{o}N, \mathbf{s}N)$ *where* $\mathbf{i}N$ *and* $\mathbf{o}N$ *are sets of ports,*

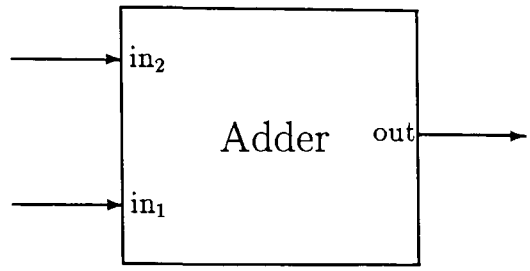


Figure 4.1: Diagrammatic Representation of a System

the inports and the outports respectively, and $\mathbf{s}N$ *is a mapping from ports to sorts. An interface must satisfy the following conditions:*

1. $\mathbf{i}N \cap \mathbf{o}N = \emptyset$,

2. $domain(\mathbf{s}N) \subseteq (\mathbf{i}N \cup \mathbf{o}N)$.

We can use **i**, **o** and **s** as operators on interfaces. For example, given an interface N then **i** applied to N, $\mathbf{i}N$, will return the set of inports of N.

We will represent systems and their interfaces pictorially using diagrams like the one shown in figure 4.1.

Example 4.1 The system represented in figure 4.1 is named Adder, and has interface

$(\{\text{in}_1, \text{in}_2\}, \{\text{out}\}, \{\text{in}_1 \mapsto \{0,1\}, \text{in}_2 \mapsto \{0,1\}, \text{out} \mapsto \{0,1\}\})$.

An interaction is the passage of an object from the system to the environment, or vice-versa, through a particular port. We will, therefore, model each interaction by a dotted pair; for example, $m.o$, where m is the port at which the interaction takes place and o is the object being passed through the port, for an interface N, where $o \in \mathbf{s}N(m)$. In order to simplify our model we assume that interactions are instantaneous and only one interaction takes place at any one time. We will ignore the interval of time seperating interactions.

Definition 4.2 *The* alphabet *of a system S is the set of all interactions in which it could participate, and we denote this by αS. Hence, $\alpha S = \{m.o : m \in \mathbf{i}N \cup \mathbf{o}N, o \in \mathbf{s}N(m)\}$ where S has interface N.*

Definition 4.3 *The* trace *of a system is a recording of its behaviour as the sequence of interactions in which it has participated up to that moment. Initially we take the trace to be empty. The trace of a system is an element of αS^*.*

Let us consider a system S which has participated in some trace t and may now take part in some interaction from the set Y. Its environment E, which has also participated in t, is now able to take part in any interaction from some set X. We might assume that S and E would take part in an interaction from $X \cap Y$. This is too strong. We say that *if*

S takes part in an interaction it will take part in one from $X \cap Y$, it may, however, take part in no interactions and so will 'halt', that is do never do anything ever again. In this case, if S 'refuses' to interact with its environment, X is a *refusal* of S after t. After a different trace X need not be a refusal. Refusals are of interest when considering system behaviour because we are not only concerned with what a system will do but with what it will not do. This allows us to deal properly with non-determinism because an interaction can both a valid next interaction for the system after some trace and an element of a refusal after the same trace. The 'choice' between whether to take part in the interaction or 'halt' is considered to be non-deterministic. Though αE is not necessarily the same as αS since we are interested in S we will define a refusal of a system S to be a subset of αS.

We wish to record the behaviour of a system as a pair of a trace and a refusal. If the system after a trace t does not refuse the set offered by the environment we will, by convention, denote this by $t, \emptyset$. If the system does refuse a set X we will denote this by t, X.

Definition 4.4 *A* behaviour *of a system S is a pair (t, X) with a trace $t \in \alpha S^*$ and a refusal $X \subseteq \alpha S$.*

Definition 4.5 *A (behavioural)* specification *of a system S is a predicate P over $\alpha S^* \times \alpha S$, and, given $t \in \alpha S^*$ and $X \subseteq \alpha S$, $P(t, X)$ means t is an acceptable trace of S and*

that X is an acceptable refusal of S after t. That is (t, X) is an acceptable behaviour of S.

We choose this approach to specifying and modelling system behaviour because it is abstract, captures many of the properties of 'real' system behaviour and is fairly straightforward to use to define our reliability concepts in a way that can be related to the 'real' behaviour of systems. This approach gives some insight into the concept of state, in particular it helps us recognise that state is an external phenomenon. That is, state is part of our perception of a system it is not intrinsic to the system. We will come back to state towards the end of this chapter.

A specification P must have the following properties, reflecting natural properties of the behaviour of systems:

1. $P(\langle\rangle, \emptyset)$, the system has done nothing,
2. $P(t, X) \Rightarrow \forall Y \subseteq X : P(t, Y)$, if the system can refuse X after t it can refuse any subset of X after t,
3. $P(t, \emptyset) \Rightarrow \forall r \leq t : P(r, \emptyset)$, if the system can participate in t it must be able to participate in any prefix of t,
4. $\forall x \in \alpha S : P(t, X) \Rightarrow P(t \wedge \langle x \rangle, \emptyset) \vee P(t, X \cup \{x\})$, if x is a possible interaction for the system, it must either be able to participate in it or refuse to participate in it.

Example 4.2 The following is a specification of the system, Adder, of figure 4.1, which adds two numbers:

$$
\begin{array}{ll}
P(\langle\rangle, R) & \text{if} \quad R \subseteq \{\text{out}.1, \text{out}.2\}, \\
P(\langle \text{in}_k.x \rangle, R) & \text{if} \quad R \subseteq \{\text{out}.i, \text{in}_k.j : i,j \in \{0,1\}\}, x \in \{0,1\}, k = 1,2, \\
P(\langle \text{in}_1.x, \text{in}_2.y \rangle, R) & \text{if} \quad R \subseteq \{\text{in}_1.i, \text{in}_2.j : i,j \in \{0,1\}\}, x,y \in \{0,1\}, \\
P(\langle \text{in}_2.x, \text{in}_1.y \rangle, R) & \text{if} \quad R \subseteq \{\text{in}_1.i, \text{in}_2.j : i,j \in \{0,1\}\}, x,y \in \{0,1\}, \\
P(\langle \text{in}_1.x, \text{in}_2.y, \text{out}.x+y \rangle, R) & \text{if} \quad R \subseteq \{\text{in}_1.i, \text{in}_2.j : i,j \in \{0,1\}\}, x,y \in \{0,1\}, \\
P(\langle \text{in}_2.x, \text{in}_1.y, \text{out}.x+y \rangle, R) & \text{if} \quad R \subseteq \{\text{in}_1.i, \text{in}_2.j, \text{out}.k : i,j,k \in \{0,1\}\}, x,y \in \{0,1\}.
\end{array}
$$

We do not recommend our modelling approach as a specification technique. However trace based methods are useful in the specification of communicating systems[14].

Definition 4.6 *We say a system S satisfies a specification P if and only if for all behaviours (t, X) of S, $P(t,X)$. We abbreviate S satisfies the specification P to S* **sat** *P. The following will hold concerning this relation:*

1. S **sat** true,

2. S **sat** $(\forall i : P_i)$ *if* $(\forall i : S$ **sat** $P_i)$

3. S **sat** Q *if* $(S$ **sat** $P) \wedge (P \Rightarrow Q)$

Let us consider example 4.1, if a number is input to the system it does not cease to be a

system, so its future behaviour must obey some specification.

Definition 4.7 *The operation* **after** *is defined by the following (where s is a trace):* $(S \textbf{ after } s)$ **sat** $P(t, X)$ *if and only if* $(\exists Q : P(t, X) = Q(s \wedge t, X) \wedge (S \textbf{ sat } Q))$. S **after** s *has the same interface as* S.

In the previous definition we should also show that if Q has the four properties we require of a specification then P will also have those four properties. This is required to show that our definitions are valid. It is simple to demonstrate but to do this for all our definitions would greatly lengthen them. So we will omit these proofs.

Example 4.3 $(Adder \textbf{ after } \langle \mathrm{in}_1.1 \rangle) \textbf{ sat } Q$ where

$Q(\langle\rangle, R)$ if $R \subseteq \{\mathrm{in}_1.i, \mathrm{out}.j : i, j \in \{0,1\}\}$,

$Q(\langle \mathrm{in}_2.x \rangle, R)$ if $R \subseteq \{\mathrm{in}_1.i, \mathrm{in}_2.j : i, j \in \{0,1\}\}, x \in \{0,1\}$,

$Q(\langle \mathrm{in}_2.x, \mathrm{out}.x+1 \rangle, R)$ if $R \subseteq \{\mathrm{in}_1.i, \mathrm{in}_2.j, \mathrm{out}.k : i, j, k \in \{0,1\}\}, x, y \in \{0,1\}$.

We still have to consider in_1, because it is still part of the interface of *Adder*.

We will define a number of "operations" on systems. These operations do not change the system but only our view of it. We will define these changes of view, i.e. operations, by considering the change in the interface and what specification our new view of the system satisfies.

Consider the two specifications given below:

$$
\begin{array}{ll}
Q(\langle\rangle, R) & \text{if} \quad R \subseteq \{\text{out}.i : i \in \{0,1\}\}, \\
Q(\langle \text{in}.x \rangle, R) & \text{if} \quad R \subseteq \{\text{in}.i : i \in \{0,1\}\}, x \in \{0,1\}, \\
Q(\langle \text{in}.x, \text{out}.x+1 \rangle, R) & \text{if} \quad R \subseteq \{\text{in}.i, \text{out}.j : i,j \in \{0,1\}\}, x,y \in \{0,1\},
\end{array}
$$

$$
\begin{array}{ll}
P(\langle\rangle, R) & \text{if} \quad R \subseteq \{\text{put}.i : i \in \{0,1\}\}, \\
P(\langle \text{get}.x \rangle, R) & \text{if} \quad R \subseteq \{\text{get}.i : i \in \{0,1\}\}, x \in \{0,1\}, \\
P(\langle \text{get}.x, \text{put}.x+1 \rangle, R) & \text{if} \quad R \subseteq \{\text{get}.i, \text{put}.j : i,j \in \{0,1\}\}, x,y \in \{0,1\}.
\end{array}
$$

These could be different views of the same system; the difference being that we have changed the names of the ports.

In the definitions and proofs that follow t, t', t'', ..., represent traces, X, X', X'', ..., represent refusals and P, Q, R, ..., represent specifications.

Definition 4.8 *Let S be a system with interface N, and let $x \in \mathbf{i}N \cup \mathbf{o}N$ and $y \notin \mathbf{i}N \cup \mathbf{o}N$. We define S* **ren** *x, y, S with x* renamed as *y, by the following, (let N' denote the interface of S* **ren** *x, y)*

$$
\mathbf{i}N' = \begin{cases} (\mathbf{i}N \setminus \{x\}) \cup \{y\} & \text{if } x \in \mathbf{i}N \\ \mathbf{i}N & \text{otherwise} \end{cases}
$$

$$
\mathbf{o}N' = \begin{cases} (\mathbf{o}N \setminus \{x\}) \cup \{y\} & \text{if } x \in \mathbf{o}N \\ \mathbf{o}N & \text{otherwise} \end{cases}
$$

$$
\mathbf{s}N'(z) = \begin{cases} \mathbf{s}N(x) & \text{if } z = y \\ \mathbf{s}N(z) & \text{otherwise} \end{cases}
$$

and $(S$ **ren** $x, y)$ **sat** $P(t, X)$ *if and only if*

$\exists Q, t', X' : (t' = f \star t) \wedge (X' = f \star X) \wedge (Q(t', X') = P(t, X)) \wedge (S$ **sat** $Q(t', X'))$

where

$$f(z.a) = \begin{cases} x.a & \text{if } z = y \\ z.a & \text{otherwise} \end{cases}$$

The following lemma states that the future behaviour of the system is independent of the names of the ports in the systems previous behaviour.

Lemma 4.1 $((S$ **ren** $x, y)$ **after** $s)$ **sat** $P(t, X)$

if and only if

$((S$ **after** $f \star s)$ **ren** $x, y)$ **sat** $P(t, X)$

where

$$f(z.a) = \begin{cases} x.a & \text{if } z = y \\ z.a & \text{otherwise} \end{cases}$$

Proof

$((S$ **ren** $x, y)$ **after** $s)$ **sat** $P(t, X)$

iff {definition 4.7}

$\exists Q : (P(t, X) = Q(s \wedge t, X)) \wedge (S$ **ren** $x, y)$ **sat** $Q(s \wedge t, X))$

iff {definition 4.8}

$\exists Q : (P(t, X) = Q(s \wedge t, X)) \wedge (\exists R, t', X' : (t' = f \star (s \wedge t)) \wedge (X' = f \star X) \wedge$

$(R(t', X') = Q(s \wedge t, X)) \wedge (S$ **sat** $R(t', X')))$

iff {Eliminating Q}

$\exists R, t', X' : (t' = f \star (s \wedge t)) \wedge (X' = f \star X) \wedge (R(t', X') = P(t, X)) \wedge (S \textbf{ sat } R(t', X'))$

iff $\{t' := (f \star s) \wedge t''\}$

$\exists R, t'', X' : (t'' = f \star t) \wedge (X' = f \star X) \wedge (R((f \star s) \wedge t'', X') = P(t, X)) \wedge (S \textbf{ sat } R((f \star s) \wedge t'', X'))$

iff {Introduce M: $M(t'', X') = R((f \star s) \wedge t'', X')$ }

$\exists M, t'', X' : (t'' = f \star t) \wedge (X' = f \star X) \wedge (M(t'', X') = P(t, X)) \wedge (S \textbf{ sat } M(t'', X'))$

iff {definition 4.7}

$\exists M, t'', X' : (t'' = f \star t) \wedge (X' = f \star X) \wedge (M(t'', X') = P(t, X)) \wedge ((S \textbf{ after } f \star s) \textbf{ sat } M(t'', X'))$

iff {definition 4.8}

$((S \textbf{ after } f \star s) \textbf{ ren } x, y) \textbf{ sat } P(t, X)$ □

4.2 Abstraction

We are now going to consider three forms of behavioural abstraction. Abstraction is the viewing of behaviour in a different way; this often leads to a loss of information and detail, but a reduction in clutter can lead to greater understanding.

4.2.1 Hiding

The first form of abstraction is the most elementary; we ignore completely those ports we are not interested in. Let $C \subseteq \mathbf{i}N \cup \mathbf{o}N$ then we write S **hide** C to indicate the system S with the ports C ignored.

Example 4.4 (*Adder* **hide** $\{\mathrm{in}_1\}$) **sat** Q where

$Q(\langle\rangle, R)$ if $R \subseteq \{\mathrm{out}.j : j \in \{0,1\}\}$,

$Q(\langle \mathrm{in}_2.x \rangle, R)$ if $R \subseteq \{\mathrm{in}_2.j : j \in \{0,1\}\}, x \in \{0,1\}$,

$Q(\langle \mathrm{in}_2.x, \mathrm{out}.y \rangle, R)$ if $R \subseteq \{\mathrm{in}_2.i, \mathrm{out}.j : i,j \in \{0,1\}\}, x,y \in \{0,1\}$.

Consider a system S which satisfies the specification P given by:

$P(\langle\rangle, \emptyset)$,

$P(\langle\rangle, \{a\})$,

$P(x, \emptyset)$ if $x \in \{c\}^*$ and

$P(x \wedge \langle a \rangle, X)$ if $x \in \{c\}^* \wedge X \subseteq \{a, c\}$.

What specification does S **hide** $\{c\}$ satisfy? This gives rise to a problem because we are ignoring possibly infinite sequences of interactions. We will include the simplification that we cannot ignore potentially unbounded sequences of communication. That is, we cannot hide ports on which such sequences can occur. How to deal with the problem of unbounded sequences of communication in trace theory was dealt with by Brookes and Roscoe[14].

Definition 4.9 *Let S be a system with interface N, and let C be a set of ports with $C \subseteq \mathbf{i}N \cup \mathbf{o}N$, then S* **hide** *C has interface $(\mathbf{i}N \setminus C, \mathbf{o}N \setminus C, \mathbf{s}N)$ and*

$(S$ **hide** *$C)$* **sat** *$P(t, X)$ if and only if*

$\exists Q, t', X' : (t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (Q(t', X') = P(t, X)) \wedge (S \textbf{ sat } Q(t', X')))$ *where* $b(x) = (x \in C)$.

The next lemma shows that the future behaviour of the system is independent of what was ignored in the past.

Lemma 4.2 $((S$ **after** $s)$ **hide** $C)$ **sat** $P(t, X)$ *iff*

$((S$ **hide** $C)$ **after** $b \triangleright s)$ **sat** $P(t, X)$

where $b(x) = (x \in C)$.

Proof

$((S$ **hide** $C)$ **after** $b \triangleright s)$ **sat** $P(t, X)$

iff {definition 4.7}

$\exists Q : (P(t, X) = Q((b \triangleright s) \wedge t, X)) \wedge ((S \textbf{ hide } C) \textbf{ sat } Q((b \triangleright s) \wedge t, X))$

iff {definition 4.9}

$\exists Q : (P(t, X) = Q((b \triangleright s) \wedge t, X)) \wedge (\exists R, t', X' : ((b \triangleright s) \wedge t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (R(t', X') =$

$Q((b \triangleright s) \wedge t, X)) \wedge (S \textbf{ sat } R(t', X')))$

iff {Eliminate Q}

$\exists R, t', X' : ((b \triangleright s) \wedge t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (R(t', X') = P(t, X)) \wedge S \textbf{ sat } R(t', X'))$

iff $\{t' := s \wedge t''\}$

$\exists R, t'', X' : (t = (b \triangleright t'')) \wedge X = b \triangleright X' \wedge (R(s \wedge t'', X') = P(t, X)) \wedge (S \textbf{ sat } R(s \wedge t'', X'))$

iff {Introduce M: $M(t'', X) = R(s \wedge t'', X)$

$\exists M, R, t'', X' : (t = (b \triangleright t'')) \wedge (X = b \triangleright X') \wedge (M(t'', X) = P(t, X)) \wedge (S \textbf{ sat } M(t'', X')))$

iff {definition 4.7}

$\exists M, t'', X' : (t = b \triangleright t'') \wedge (X = b \triangleright X') \wedge (M(t'', X) = P(t, X)) \wedge ((S \textbf{ after } s) \textbf{ sat } M(t'', X'))$

iff {definition 4.9}

$((S \textbf{ after } s) \textbf{ hide } C) \textbf{ sat } P(t, X)$ □

The following lemma says that the hiding of port is the same as renaming a port and then hiding it.

Lemma 4.3 $(S \textbf{ hide } \{c\}) \textbf{ sat } P(t, X)$ *if and only if*

$((S \textbf{ ren } c, d) \textbf{ hide } \{d\}) \textbf{ sat } P(t, X)$.

Proof

Let $b(x) = x = d$, $b'(x) = x = c$,

$$f(x) = \begin{cases} c & \text{if } x = d \\ x & \text{otherwise} \end{cases}$$

$$g(x) = \begin{cases} d & \text{if } x = c \\ x & \text{otherwise} \end{cases}$$

$((S \textbf{ ren } c, d) \textbf{ hide } \{d\}) \textbf{ sat } P(t, x)$

iff {definition 4.9}

$\exists Q, t', X' : (t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (Q(t', X') = P(t, X)) \wedge ((S \textbf{ ren } c, d) \textbf{ sat } Q(t', X'))$

iff {definition 4.8}

$\exists Q, t', X' : (t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (Q(t', X') = P(t, X)) \wedge (\exists R, t'', X'' : (t'' = f \star t') \wedge$

$(X'' = f \star X') \wedge (Q(t', X') = R(t'', X'')) \wedge (S \textbf{ sat } R(t'', X'')))$

iff { f is invertible}

$\exists R, t'', X'' : (t = b \triangleright (g \star t'')) \wedge (X = b \triangleright (g \star X'')) \wedge (R(t'', X'') = P(t, X)) \wedge (S \textbf{ sat } R(t'', X''))$

iff { $b \triangleright (g \star s) = b' \triangleright s$}

$\exists R, t'', X'' : (t = b' \triangleright t'') \wedge (X = b \triangleright X'') \wedge (R(t'', X'') = P(t, X)) \wedge (S \textbf{ sat } R(t'', X''))$

iff { definition 4.9}

$(S \textbf{ hide } \{c\}) \textbf{ sat } P(t, X)$ □

4.2.2 Resorting

The second form of abstraction we consider is to change how we view the objects passing through the interface; for example, an eight-bit outport (i.e. one producing eight-bit numbers) can be regarded as a port producing characters. Let S be a system with interface N and let $c \in \mathbf{i}N \cup \mathbf{o}N$ and f be a function from $\mathbf{s}N(c)$ to some sort, we write $(S \textbf{ res } c, f)$ to indicate the system S with the port c, of N, abstracted by the application of f to the objects passing through c.

Example 4.5 Xor = $((Adder \textbf{ res } \mathrm{in}_1, \mathrm{truth}) \textbf{ res } \mathrm{in}_2, \mathrm{truth}) \textbf{ res } \mathrm{out}, \mathrm{truth}$ where

$truth(0) = \textbf{false}$

$truth(1) = \mathbf{true}$

Definition 4.10 *Let S be a system with interface N, let c be a port and f a function from $\mathbf{s}N(c)$ to some sort U, then S* **res** *c, f has interface $(\mathbf{i}N, \mathbf{o}N, \mathbf{s}N \cup \{c \mapsto U\})$ and S* **res** *c, f* **sat** *$P(t, X)$ if and only if $\exists Q, t', X' : (t = g \star t') \wedge (X = g \star X') \wedge$ $((Q(t', X') = P(t, X)) \Rightarrow (S \textbf{ sat } Q(t', X')))$, where*

$$g(z.a) = \begin{cases} c.f(a) & \text{if } z = c \\ z.a & \text{otherwise} \end{cases}$$

Lemma 4.4 $((S \textbf{ after } s) \textbf{ res } c, f) \textbf{ sat } P(t, X)$ *iff* $(S \textbf{ res } c, f \textbf{ after } g \star s) \textbf{ sat } P(t, X)$

where

$$g(z.a) = \begin{cases} c.f(a) & \text{if } z = c \\ z.a & \text{otherwise} \end{cases}$$

Proof

$(S \textbf{ res } c, f \textbf{ after } g \star s) \textbf{ sat } P(t, X)$

iff {By definition 4.7}

$\exists Q : (P(t, X) = Q((g \star s) \wedge t, X)) \wedge (S \textbf{ res } c, f) \textbf{ sat } Q((g \star s) \wedge t, X)$

iff {By definition 4.10}

$\exists R, t', X' : (((g \star s) \wedge t) = g \star t') \wedge (X = g \star X') \wedge (R(t', X') = P(t, X)) \wedge (S \textbf{ sat } R(t', X'))$

iff $\{t' := s \wedge t''\}$

$\exists R, t'', X' : (t = g \star t'') \wedge (X = g \star X') \wedge (R(s \wedge t'', X') = P(t, X)) \wedge S$ **sat** $R(s \wedge t'', X')$

iff {Introduce M}

$\exists M, R, t'', X' : (t = g \star t'') \wedge (X = g \star X') \wedge (M(t'', X') = R(s \wedge t'', X') = P(t, X)) \wedge$

$(S$ **sat** $R(s \wedge t'', X'))$

iff {Eliminate R}

$\exists M, t'', X' : (t = g \star t'') \wedge (X = g \star X') \wedge (M(t'', X') = P(t, X)) \wedge ((S$ **after** $s)$ **sat** $M(t'', X'))$

iff {By definition 4.10}

$(S$ **after** $s)$ **res** c, f **sat** $P(t, X)$ □

The following lemma shows the independence of the rename and resort abstractions.

Lemma 4.5 $((S$ **res** $c, f)$ **ren** $c, d)$ **sat** $P(t, X)$ *iff* $((S$ **ren** $c, d)$ **res** $d, f)$ **sat** $P(t, X)$

Proof

Let

$$g(z.a) = \begin{cases} c.a & \text{if } z = d \\ z.a & \text{otherwise} \end{cases}$$

$$g'(z.a) = \begin{cases} d.a & \text{if } z = c \\ z.a & \text{otherwise} \end{cases}$$

$$h(z.a) = \begin{cases} c.f(a) & \text{if } z = c \\ z.a & \text{otherwise} \end{cases}$$

$$h'(z.a) = \begin{cases} d.f(a) & \text{if } z = d \\ z.a & \text{otherwise} \end{cases}$$

$((S \textbf{ res } c, f) \textbf{ ren } c, d) \textbf{ sat } P(t, X)$

iff {By definition 4.8}

$\exists Q, t', X' : (t' = g \star t) \wedge (X' = g \star X) \wedge (Q(t', X') = P(t, X)) \wedge ((S \textbf{ res } c, f) \textbf{ sat } Q(t', X'))$

iff {By definition 4.10}

$\exists Q, t', X' : (t' = g \star t) \wedge (X' = g \star X) \wedge Q(t', X') = P(t, X) \wedge (\exists R, t'', X'' : (t' = h \star t'') \wedge$

$(X' = h \star X'') \wedge (Q(t', X') = R(t'', X'')) \wedge (S \textbf{ sat } R(t'', X'')))$

iff {Eliminate Q}

$\exists R, t'', X'' : (g \star t = h \star t'') \wedge (g \star X = h \star X'') \wedge (P(t, X) = R(t'', X'')) \wedge (S \textbf{ sat } R(t'', X''))$

iff {g is invertible}

$\exists R, t'', X'' : (t = (g' \circ h) \star t'') \wedge (X = (g' \circ h) \star X'') \wedge (P(t, X) = R(t'', X'')) \wedge (S \textbf{ sat } R(t'', X''))$

iff {$g' \circ h = h' \circ g$}

$\exists R, t'', X'' : (t = (h' \circ g) \star t'') \wedge (X = (h' \circ g) \star X'') \wedge (P(t, X) = R(t'', X'')) \wedge (S \textbf{ sat } R(t'', X''))$

iff {Introduce M, t' and X'}

$\exists M, R, t'', X'', t', X' : (t = h' \star t') \wedge (X = h' \star X') \wedge (t' = g \star t'') \wedge (X' = g \star X'') \wedge$

$(P(t, X) = M(t', X') = R(t'', X'')) \wedge (S \textbf{ sat } R(t'', X''))$

iff {By definition 4.8}

$\exists M, t', X' : (t = h' \star t') \wedge (X = h' \star X') \wedge (P(t, X) = M(t', X')) \wedge ((S \textbf{ ren } c, d) \textbf{ sat } M(t', X'))$

iff {By definition 4.10}

$((S \textbf{ ren } c, d) \textbf{ res } d, f) \textbf{ sat } P(t, X)$ □

We now show that changing the way we view interactions at a port and then ignoring that port is equivalent to just ignoring the port.

Lemma 4.6 $((S \textbf{ res } c, f) \textbf{ hide } \{c\}) \textbf{ sat } P(t, X)$ *if and only if* $(S \textbf{ hide } \{c\}) \textbf{ sat } P(t, X)$

Proof

Let $b(x) = x = c$ and

$$g(z.a) = \begin{cases} c.f(a) & \text{if } z = c \\ z.a & \text{otherwise} \end{cases}$$

$((S \textbf{ res } c, f) \textbf{ hide } \{c\}) \textbf{ sat } P(t, X)$

iff {By definition 4.9}

$\exists Q, t', X' : (t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (Q(t', X') = P(t, X)) \wedge ((S \textbf{ res } c, f) \textbf{ sat } P(t, X))$

iff {By defintion 4.10}

$\exists Q, t', X' : (t = b \triangleright t') \wedge (X = b \triangleright X') \wedge (Q(t', X') = P(t, X)) \wedge$

$(\exists R, t'', X'' : (t' = g \star t'') \wedge (X' = g \star X'') \wedge (R(t'', X'') = Q(t', X')) \wedge (S \textbf{ sat } R(t'', X'')))$

iff {Eliminate Q}

$\exists R, t'', X'' : (t = b \triangleright g \star t'') \wedge (X = b \triangleright (g \star X'')) \wedge (R(t'', X'') = P(t, X)) \wedge (S \textbf{ sat } R(t'', X''))$

iff { $b \triangleright (g \star t) = b \triangleright t$}

$\exists R, t'', X'' : (t = b \triangleright t'') \wedge (X = b \triangleright X'') \wedge (R(t'', X'') = P(t, X)) \wedge (S \textbf{ sat } R(t'', X''))$

iff {By definition 4.9}

$(S \textbf{ hide } \{c\}) \textbf{ sat } P(t, X)$ □

4.2.3 Collapsing

The final method of abstraction we will deal with is the collapsing of a group of interactions on a number of ports to an interaction on one port; for example, viewing eight one-bit ports as one eight-bit port. We will refer to each group of interactions that are to be collapsed down to a single interaction as an action.

Example 4.6 Let us consider the *Adder* of figure 4.1. We wish to view this system as one which inputs a two bit number and produces a parity bit by adding the bits of the number together. It has interface $I = (\{\text{in}\}, \{\text{out}\}, \{\text{in} \mapsto \{00, 01, 10, 11\}, \text{out} \mapsto \{0, 1\}\})$.

$Q(\langle\rangle, R)$ if $R \subseteq \{\text{in}.i : i \in \mathbf{s}I(\text{in})\}$

$Q(\langle \text{in}.i \rangle, R)$ if $i \in \mathbf{s}I(\text{in}) \wedge R \subseteq \{\text{out}.j : j \neq \text{sum}(i)\}$

$Q(\langle \text{in}.i, \text{out}.\text{sum}(i) \rangle, \emptyset)$ if $i \in \mathbf{s}I(\text{in})$

where

$\text{sum}(00) = 0$

$\text{sum}(01) = 1$

$\text{sum}(10) = 1$

$\text{sum}(11) = 0$

An action is a group of interactions. In the above example we have $\langle \text{in}_1.0, \text{in}_2.0 \rangle \mapsto \langle \text{in}.0 \rangle$ and $\langle \text{in}_2.0, \text{in}_1.0 \rangle \mapsto \langle \text{in}.0 \rangle$. So we have a pair of interactions which make up an action but this action has two "forms". By "forms" we mean the allowable sequences of interactions

that make up an individual action. We will formally define forms, but first let us define the merging of traces.

The ordered merge of two traces is the set of traces in which each of the traces in the set contains the same bag of interactions as the two traces and all the interactions on any given port in all of the traces in the set are in the order they would have been in had the first trace followed the second trace.

Definition 4.11 *The* ordered merge *of two traces r and s is the set of traces:*

$\{t : (t = r_0 \wedge s_1 \wedge r_1 \ldots r_{n-1} \wedge s_n \wedge r_n) \wedge$

$(r_0 \wedge \ldots \wedge r_n = r) \wedge (s_1 \wedge \ldots \wedge s_n = s) \wedge (r_0 \neq \langle\rangle) \wedge$

$(\forall$ *ports* $c : (b(c.a) = a) \wedge b \triangleright t = b \triangleright (r \wedge s))\}$.

The ordered merge of a sequence of traces is carried out by taking the first two traces, merging them, merging the result with the next trace in the sequence and so on. The ordered merge of a sequence of traces is written $merge\langle t_1, \ldots, t_n \rangle$.

Definition 4.12 *Let S be a system with interface I, $A \subseteq (\mathbf{i}I \cup \mathbf{o}I)$ and F a set of traces. The ports used in F must be a subset of A.*

(A, F) *is the* forms *of an action on S if and only if* $\forall t \in merge\langle p_1, \ldots, p_m, u_1, \ldots, u_n \rangle$:

$(p_i \in F) \wedge (u_j \in \{t : t < s, s \in F\}) \wedge t$ *is a valid trace of* $(S$ **hide** $((\mathbf{i}I \cup \mathbf{o}I) \setminus A))$

($<$ is the prefix relation on traces)

When writing down a form we will often only write down the sequence of interactions, since the set of ports involved is then obvious.

Example 4.7 The forms of the action in the previous example are:

$\{\langle \mathrm{in}_1.0, \mathrm{in}_2.0\rangle, \langle \mathrm{in}_1.1, \mathrm{in}_2.0\rangle, \langle \mathrm{in}_1.0, \mathrm{in}_2.1\rangle, \langle \mathrm{in}_1.1, \mathrm{in}_2.1\rangle,$

$\langle \mathrm{in}_2.0, \mathrm{in}_1.0\rangle, \langle \mathrm{in}_2.1, \mathrm{in}_1.0\rangle, \langle \mathrm{in}_2.0, \mathrm{in}_1.1\rangle, \langle \mathrm{in}_2.1, \mathrm{in}_1.1\rangle\}$

Why have we defined the forms of an action in this way? It means that actions can overlap but we can say of any two actions that the last interaction of the first action will occur before the last interaction of the second action. So not only do actions start in some order they also finish in the same order. The restriction is stronger than just this because we allow an action to have an interaction on a particular port only after all the actions which started earlier have ceased to use that port.

Now that we have considered what groups of interactions form valid actions we move on to consider how to replace these actions with a single interaction.

Example 4.8 Let S **sat** Q where

$Q(\langle\rangle, \{b.0\})$

$Q(\langle a.0\rangle, \{a.0, c.0\})$

$Q(\langle c.0\rangle, \{a.0, c.0\})$

$Q(\langle a.0, b.0\rangle, \{a.0, b.0\})$

$Q(\langle c.0, b.0\rangle, \{b.0, c.0\})$

$Q(\langle a.0, b.0, c.0\rangle, \{a.0, b.0, c.0\})$

$Q(\langle c.0, b.0, a.0\rangle, \{a.0, b.0, c.0\})$

and let us consider the forms of an action on S $\{\langle a.0, c.0\rangle, \langle c.0, a.0\rangle\}$.

Looking at example 4.8 we can see that we have valid forms for an action. What we wish to do is collapse the action defined by this form to a new interaction $d.0$ (say). That is we wish to replace $a.0$ and $c.0$ together by $d.0$. But we cannot collapse this action to a single interaction because $b.0$ must occur between $a.0$ and $c.0$. If $a.0$ and $c.0$ were collapsed to a single interaction then $b.0$ would have to occur simultaneously with this new interaction. This is not possible because, by our original assumptions, we only allow one interaction to occur at a time. Therefore collapsing can only have meaning, within our trace based model, if there are no interactions outside the action which *have to* occur "during" the action. This is, of course, atomicity. So, we define an atomicity criteria.

Informally the atomicity criteria is: Given a set of ports on which the actions are to occur, and a set of allowable forms of those actions, the actions will be atomic, if and only if, for every valid trace of the system there exists another valid trace which contains the same interactions but is of the form - ("non-action interactions" $\wedge$ "action interactions")... $\wedge$ "interactions, but only unfinished actions", and the system has the same future behaviour "after" either trace.

This leads us to the following definition.

Definition 4.13 *Let S be a system with interface N and let P be a specification with S* **sat** *P. Let A be a set of ports of S, with either $A \subseteq \mathbf{i}N$ or $A \subseteq \mathbf{o}N$, and let (A, F) be the forms of an action on S then we say the actions on A defined by F are atomic if and only if*

$(\forall t, X : P(t, X) \Rightarrow (\exists n_i \in (\alpha S \setminus A)^*, p_i \in F, u_i \in t : t < s, s \in F :$

$(t \in merge\langle n_0, p_1, n_1 \ldots, p_k, n_k, u_1, \ldots, u_{m-k}, n_m\rangle)\wedge$

$(\forall s, Y : P(n_0 \wedge p_1 \wedge n_1 \wedge \ldots \wedge p_k \wedge n_k \wedge u_1 \ldots \wedge u_{m-k} \wedge n_m \wedge s, Y) = P(t \wedge s, Y))\wedge$

$b \triangleright t = n_1 \wedge \ldots \wedge n_m))$

where

$b(d.a) = d \in A.$

We have a further problem to consider concerning collapsing actions; what happens if during an action the system can refuse to finish it; i.e. the system can refuse the next interaction in the sequence (as defined by the forms of the action)? We will take this to mean that the system can refuse the whole action, and when we come to define our collapsing we will say that the system can refuse the interaction the action collapses to.

We will write S **col** A, F, c, f to indicate that interactions on the ports A as defined by the forms (A, F) are collapsed to interactions on the port c by way of f. That is we will take the actions defined by F which are behaviours of S and for each of these map them

onto an interaction on the port c using the function f, which is a mapping from F to some sort.

Definition 4.14 *Let S be a system with interface N. Let A be a set of ports of S, let (A, F) be the forms of an action on S with the actions defined atomic and let c be a port and f be a function with $F \subseteq domain(f)$.*

$(S$ **col** $A, F, c, f)$ **sat** $P(t, X)$ *if there exists* $s, Y, n_0, \ldots, n_m, p_1 \ldots p_k, u_1, \ldots, u_{m-k}$

$n_i \notin F, p_i \in F, u_i < r_i where r_i \in F$

$t = n_0 \wedge c.f(p_1) \ldots \wedge c.f(p_k) \wedge n_k,$

$X = b \triangleright Y \cup \{c.f(r) : ((u_i \wedge \langle y \rangle) \leq r) \wedge (r \in F) \wedge (y \in Y)\},$

$b(d.a) = d \in A,$

$s \in merge\langle n_0, p_1, \ldots, p_k, n_k, u_1, \ldots, u_{m-k}, n_m \rangle$

$(\forall v, Z : Q(s \wedge v, Z) = Q(n_0 \wedge p_1 \ldots \wedge p_k \wedge n_k \wedge u_1 \ldots \wedge u_{m-k} \wedge n_m \wedge v, Z))$ *and*

$P(t, X) = Q(s, Y)$ *and*

S **sat** $Q(s, Y)$.

The concept of collapsing requires more work. In particular, it needs to be shown that the specification produced by the collapsing operation meets our four requirements. Also, we need to relate collapsing to the other forms of abstraction.

4.3 States

Consider a box with three lights on it; a red light, a green light and a blue light. Initially the red light is on and the behaviour of the system is for the red to go off, the green to come on, the green to go off, the blue to come on, the blue to go off, then the green to come on, etc. The lights cycle through red, green, blue, green, red, green, blue, green, ... If the green light is on we cannot deduce from the specification and the current output alone whether the red or the blue light will come on next. We need some additional piece of information. Loosely, we call this piece of information the *state* of the system.

For our box we could describe four states: red, blue, green going to blue and green going to red. Or, alternatively, red, blue, green after red, and green after blue.

So, the state of a system S is any amount of information, extra to its specification, which will allow us to predict the future behaviour of S. This state is not directly observable by the environment but is deduced from the systems behaviour. Given a system we need to know both its specification and its state in order to be able to predict its future behaviour.

Our choice of how we represent the state will vary according to the circumstances.

Example 4.9 We can represent the state of a simple four-bit memory chip as a state table:

Addr	Data
00	0001
01	0010
10	0100
11	1000

or as a function:

$$\text{data}(\text{addr}) = 2^{\text{addr}}$$

Definition 4.15 *Let S be a system and P a specification with S* **sat** *$P(t, X)$, the* minimal set of states *of S is any set isomorphic to the set*

$$\{\{(r, X) : P(s \wedge r, X)\} : P(s, Y)\};$$

that is the set of sets of future behaviours.

We choose future behaviours for our definition in preference to past behaviours because it allows us to deal with non-deterministic systems and lets us have the idea of hidden state changes. It can also be seen from this definition that the states of a system are defined with respect to [a specification.

The system takes part in a series of interactions and as it interacts its state may change. Such state transitions are easily deducable by the environment since it can observe the

interaction. However a system could be designed such that if one presses a button within two seconds it flashes a green light and if one takes more than two seconds it flashes a red light. The future behaviour of the system is not dependent only upon interactions with the environment, i.e. pressing the button, the system may also go through state changes, which are not as immediately obvious. We refer to such state changes as *hidden state changes.*

Example 4.10 Let us say that our box is non-deterministic in going from green to red or blue. That is we do not know whether it is in the state green going to blue or green going to red until a output appears.

Definition 4.16 *Let S be a system and P a specification with S* **sat** *$P(t, X)$ The set of states of S is any set for which there exists a function whose image is the whole set*

$$\{\{(r, Y) : P(s \wedge r, X)\} : P(s, Y)\}.$$

Because we can choose any representation for the state, we could choose one that was structured, according to our definition of chapter 3. The structure we choose for our state is often based on our knowledge of the structure of the system. This relationship between the structure of the state and the structure of the system is probably the cause of the belief that the state of the system is in some way internal to the system.

4.3.1 Internal State

When we are interacting with a system we have some idea of its state. We call this the *external state* of a system. We can also define the *internal state* of a system at a level of detail to be the ordered set of external states of its components given by that level of detail. Thus, the internal state at the 'highest' level of detail (that is the one containing the lowest amount of detail and so representing the system as a whole) is the same as the external state of the system.

There will exist a mapping which will allow us to deduce the state of the system at some level of detail from the state at some lower level of detail, for example we can deduce the external state of the system from an internal state. This mapping is of course dependent on how the system is believed to work. This means that in fact for any level of detail there are two ways of deducing the state. One is an inward looking approach from the specification of the system and its past behaviour and the second an outward approach mapping the state from a lower level of detail. These two approaches will give the same result, assuming the mappings are correctly defined, except in the presence of failures. We will discuss this again in the next chapter.

As the components function and produce outputs we will get state changes which are visible (i.e. an interaction takes place) from the current level of detail but which may not be visible at a higher level of detail (i.e. there is no observable interaction). However this

internal state may affect the external state, this is a hidden state change.

It will also be noticed that the time granularity for the level of detail is ordered in the reverse of the level of detail. That is interactions and state changes happen more frequently at lower level of detail than at higher. This is because interactions and state changes can only occur at a higher level of detail if an interaction and state change occurs at a lower level of detail, but not vice-versa.

4.3.2 Programs and Interpreters

We have been given a specification Q and we wish to build a system which meets that specification. If we have a system S we could attempt to construct a trace s such that $(S \textbf{ after } s) \textbf{ sat } Q$. If we manage to construct such a trace we say we have programmed S using s.

Consider the state of S **after** s, we can choose a representation of the states of S such that we can distinguish some part of the state which is fixed for future behaviours. This corresponds to Kopetz's i-state and we refer to it as the program. The other part of the state corresponds to Kopetz's h-state.

Having programmed S using s to meet some specification Q we could then program it using t to meet some specification P, i.e. $(S \textbf{ after } s) \textbf{ after } t$. We can then distinguish

in the state a hierarchy of programs each dependant in some way on a previously entered program. We call such programs interpreters.

The hierarchies and how they interelate are products of our model of system state and how we understand the execution of programs, that is how the programmable system operates. But normally when we deal with a program we consider its meaning in isolation from the machine on which it operates. That is the program has an operational meaning, through the language the program is written in and the semantics of the language, which is independent of the system interpreting the program, this has been called *abstract execution*[16].

Chapter 5

Reliability Concepts

We now use the models and concepts we have developed, in previous chapters, to give precise definitions for a number of important reliability concepts.

5.1 Failure

We define failure to be *the deviation of the system's actual behaviour from its required behaviour.* Normally from this one would say that in order to determine whether a system has failed we need to know what the required behaviour is. However, we wish to make a stronger statement than this; in order for a system to fail we must know what the required behaviour is. If we do not know what the required behaviour is the system cannot fail.

We are emphasising that failure is a subjective and not an absolute phenomenon.

We ascertain the required behaviour from a specification. A specification is something which tells us whether a particular behaviour is valid. For example, the specification of chapter four told us whether a particular behaviour, formalised as a trace and a refusal, was valid.

So, a system fails with respect to a particular specification. Different 'users' of a system may have different specifications for that system and because they have different ideas of how the system should behave will have different views of when it has failed. For computer systems there usually exists a unique specification that allows us to talk about a system failing in some absolute sense, however other types of systems, for example human based systems, rarely have any explicit specification.

We cannot discuss the failure of a system for behaviour not covered by its specification or for behaviour which is outside the control of the system.

Behaviour not dealt with by a specification cannot be said to be valid or invalid with respect to that specification. For example, when dealing with a computer system we usually only consider the stream of bits passing into and out of the system in our specification, so if, when we touch it, it gives us an electric shock we cannot say whether it has failed or not, with respect to this specification.

In the terms of our model of chapter four, we are only interested in interactions which are in the alphabet of the system. If the system takes part in interactions outside its alphabet we can come to no conclusions concerning their validity and so such interactions will be ignored. Some authors have a concept of exceptional inputs. We do not have this concept. Inputs not defined by a specification are ignored in order to consider such inputs we require a specification which covers them. This leads us to having multiple specifications. We will discuss this in more detail later in this chapter.

We can not say a system fails if it accepts (in the sense of the chapter 4) inputs outside its specification, since the inputs given to it by the environment are outside its control. Not only does a system not fail when it accepts an input outside its specification but it can not fail at any time thereafter.

There are two ways in which a system can deviate from its specification:

1. It doesn't do something it should, or

2. It does something it shouldn't.

A system fails at the moment in time when its behaviour deviates from that required. This must be emphasised, a system fails at the time the behaviour deviates, not at the time when it is observed to have deviated. In fact a system failure does not have to be observed in order to have occurred.

We can not divorce time from the concept of failure. For example, even though the system has done the right things in the right order, it may have done them at the wrong time.

Example 5.1 We could easily construct a "clock" which changed the time every 1.5 seconds. From a trace based view this clock might meet its specification. However it is obvious that it cannot meet the usual specification of a clock.

This is a limitation of the trace based approach we have used for the formalisation of our definitions. However, the definitions provide a framework and are helpful in understanding the reliability concepts we consider.

We now define failure in terms of our model of behaviour.

Definition 5.1 *Let S be a system with interface N and let P be a specification. Let S have participated in some trace $t \in \alpha S^*$, such that $P(t, \emptyset)$ is* **true***; that is t is an acceptable behaviour of S according to P, and let the environment offer some set of interactions X, with $X \subseteq \alpha S$. Then S* fails with respect to *P if and only if one of the following is true:*

1. *S refuses to interact and $\exists m.i \in X : \neg P(t, \{m.i\})$, or*

2. *S participates in some interaction $m.i \in X, m \in \alpha N$ and $\neg P(t \wedge \langle m.i \rangle, \emptyset)$.*

5.1.1 Specifications

Specifications, as we will see in this section, are complex objects, made up of sub-specifications for the many different aspects of system use and behaviour. These sub-specifications may cover behaviour in the presence of failures, performance, reliability etc. First, we consider the hierarchy of specifications needed to define the concept of acceptance and failure modes.

Different failures can have different consequences for the environment; from being almost indistinguishable from the required service to having catastrophic impact on the environment. When a system fails we need to be able to specify its behaviour after the failure. To do this we require an additional specification which specifies the behaviour of the system in the presence of such failures, even if it is 'don't care' or *stop*. We will usually have a set of such specifications one for each 'type' of failure. These specifications define not only the behaviour of the system after the failure but that after it. This will allow us to distinguish, for example, the types of two failures even though the actual failure events might be identical.

The system could then fail with respect to one of these specifications and we will require additional specifications to deal with this. These specifications will form a hierarchy of specifications with a top, the 'normal' behaviour of the system, and a bottom. If no bottom is explicitly defined we can use the specification stating that any behaviour is

acceptable (This is *true* in our model of chapter four).

Each of these specifications defines what we call a *mode of behaviour* of the system. We seperate the modes into two distinct sets called the acceptance modes and the failure modes. If the system is in an acceptance mode then it is behaving in a way which we find acceptable, if it is in a failure mode then it is behaving in a way which we find unacceptable. We associate different aspects of failure with different failure modes; for example, one aspect could be security and another safety.

This separation into acceptance and failure modes is subjective. Whether a particular mode is a failure or acceptance mode will differ according to a persons viewpoint.

Definition 5.2 *Let S be a system, let $Q_1, \ldots, Q_n, P_1, \ldots, P_m$ be specifications:*

$\forall i \in \{1 \ldots n\}, j \in \{1 \ldots m\}$,

$Q_1 \Rightarrow Q_i$

$Q_1 \Rightarrow P_j$

$P_j \Rightarrow P_m$

$Q_i \Rightarrow P_m$

$\neg(P_j \Rightarrow Q_i)$

Let there exist sets $A = \{a_1, \ldots, a_n\}$ and $E = \{e_1, \ldots, e_m\}$, with $A \cap E = \emptyset$, such that a_i is associated with Q_i and e_i is associated with P_i. We call the a_i acceptance modes *and the e_i* failure modes *of S.*

Let $a_i \in A$ be an acceptance mode of S and let (t, X) be a behaviour of S. S is in mode a_i after t, X if and only if $Q_i(t, X)$ and $(\forall j \neq i : Q_j(t, X) \Rightarrow \neg(Q_j \Rightarrow Q_i))$.

Let $e_i \in E$ be a failure mode of S and let (t, X) be a behaviour of S. S is in mode e_i after t, X if and only if $P_i(t, X)$, $(\forall j : \neg Q_j(t, X))$ and $(\forall j \neq i : P_j(t, X) \Rightarrow \neg(P_j \Rightarrow P_i))$.

When a system fails it may place objects in the environment in unacceptable states. In order to allow the environment to recover, or to limit the effects of this the system may produce some output which informs the environment of the problem and which will help in the process of limitation and recovery. This behaviour of the system is called *compensation* This output will be defined by one of the specification described above.

We use this approach when we are trying to define acceptable behaviour in the presence of failures. The term *graceful degradation* is often used to describe systems which will pass through a series of acceptance modes, each 'less' acceptable than its predecessor.

In fact it is common to split failure into three modes:

1. Minor: minimal effect on system function; for example, a spelling mistake on a prompt.

2. Major: major effect on system function; for example, system shutsdown unnecessarily.

3. Critical: failure endangers human life.

There is no problem with the definition of failure when we have this multitude of specifications. What we must be careful to do is always to stipulate which specification the system has failed with respect to. For example, when specifying graceful degradation we need to specify each of the acceptance modes of the system. So the specification of graceful degradation will contain a number of sub-specifications. The system may fail with respect to some of these sub-specifications without failing with respect to the specification for graceful degradation.

As well as properties, such as graceful degradation, there are many other aspects of a system's use and behaviour that we may wish to specify. For example, maintainability, performance etc.

In fact we may not be able to distinguish two modes of behaviour of a system except by considering such aspects. For example, a system operating in either of the modes a_2 and a_1 might produce the same result from the same input only in different lengths of time.

Another aspect of a systems behaviour is reliability and we can associate probabilities or other measures of reliability with each mode. This gives us a reliability specification, which can stipulate how often particular types of behaviour are permitted to occur.

We see from this that a full behavioural specification for a system is made up of many

specifications each dealing with a different aspect of a system's behaviour. And that a full system specification will deal with aspects of a system other than behaviour, for example, maintainability.

5.1.2 Conclusion

To conclude this section let us reiterate our first sentence: Failure is the deviation of the system's actual behaviour from its required behaviour as defined by a particular specification.

As we have discussed, these specifications will in fact be much more complex objects than we considered in chapter four. However, this simplistic view of specifications is still useful and hence we will continue to use it.

5.2 Errors

In order to discuss errors we need not only a specification which tells us how the system should behave but also something to describe how (we believe) it is actually behaving. What we need, in fact, is the 'real' or 'correct' specification of the system. That is, the specification which is always correct in predicting the systems behaviour. Of course, this specification does not, in general, exist, due to non-determinism and component failure.

Usually, however, we can at any point in time produce an approximation to it. For example, when a system fails we can construct a specification that provides a prediction of the system's future behaviour. This new specification can be constructed from the given system specification, our knowledge of the internals of the system and the fact of the system's failure. Also when a system fails is may move into a different but defined mode of behaviour, in which case we will have a specification for that mode which can be used as an approximation. It is probable that this new specification is not a correct specification of the system's behaviour.

These two specifications, the one that defines the required behaviour and the one that defines the 'actual' behaviour define two sets of states and state transitions. States of the system defined with respect to the specification of the actual behaviour can be considered to be the 'actual' states of the system. States of the system defined by the specification of required behaviour can be considered to be the 'correct' states of the system.

So, after some trace, the system fails with respect to the specification defining the required behaviour. It cannot fail with respect to the one defining the actual behaviour. Since states are defined with respect to specifications, the two specifications will define two distinct sequences of states for the system, both of the same length. There will exist a prefix which is the same for both sequences of states. However, at some point the sequences of states will diverge, possibly well in advance of the failure. At any point after that divergence the 'actual' and 'correct' state of the system will differ. When the

'actual' and 'correct' state differ we say that the state of the system is erroneous and the difference between the two states we call an *error*. A point to remember is that the erroneous state is the one defined by the specification giving the 'actual' behaviour. That is the erroneous state is the 'actual' state not the 'correct' state

In fact we need to consider the possibility that a system can be in an erroneous state even though it does not fail. What matters is that there exists a legal sequence of interactions which could take it from its current state to a failure. One has to be careful here. If a system does not meet a specification there must exist a sequence of legal interactions which take it from its initial state to a failure with respect to that specification. Does this mean that the initial state of the system can be considered to be erroneous? Not necessarily. Whether a state is erroneous depends on whether it differs from the 'correct' state and so depends upon our specification of the 'actual' behaviour and on our choice of associated states.

We have defined an error as the difference between the 'actual' state and the 'correct' state. However, we have not defined what we mean by difference nor can we. What is meant by the difference between two states must be decided by how we choose to represent those states. For example, if I talk about the state of an integer variable, then the error could be a number, but if that integer variable represents a colour I might be able to say no more than that the colour is wrong.

One can see that the choice of the representation of the state for the two specifications is

crucial if we are going to have a meaningful concept of an error for a particular system. However, it is not a great problem since our choice of representation for the state is normally based on some knowledge of the system's internal structure and behaviour.

Definition 5.3 *Let S be a system, Q and P be specifications, and let S* **sat** *Q. (P is the specification of the required behaviour of S and Q is the specification of the 'actual' behaviour of S).*

Let there exist some trace $t \in \alpha S^*$, such that $Q(t, \emptyset)$ and $P(t, \emptyset)$ are **true** *and*

$$\exists s : \neg P(t \wedge s, \emptyset) \wedge Q(t \wedge s, \emptyset) \wedge (\forall n < t \wedge s : P(n, \emptyset)))$$

Let P and Q both define the sequence of states r for S given t, let P define the sequence of states $(r \wedge \langle p_1 \ldots p_n \rangle)$ for S given $t \wedge s$ and let Q define the sequence of state $(r \wedge \langle q_1 \ldots q_n \rangle)$ for S given $t \wedge s$. If $q_i \neq p_i$ then we say that q_i is an erroneous state *of S with respect to P. The 'difference' between q_i and p_i is referred to as an* error.

An error is defined with respect to two specifications. The 'current' state of a system is erroneous with respect to those specifications if the the two states defined with respect to the two specification are different and there is a trace of interactions, valid by the specification of the required behaviour, which can lead to a failure.

We could have taken an alternative approach to defining an error. Rather than having to find a specification from which to produce the 'actual' state of the system we could have

considered the internal state of the system. We could then have used a mapping from internal states to external states. The approach taken here is more general because it does not exclude the internal state approach since the specification of the 'actual' behaviour could be obtained by looking at the internal structure and behaviour of the system. The approach adopted here also allows us tOOCo make use of other sources of knowledge, such as an understanding of the semantics of a program.

We can define an internal state of a system at a level of detail to be erroneous if at least one of the external states of the components that comprise it is erroneous. It is possible that the state of the system could be considered erroneous even though the internal states at all levels of details were not erroneous. That is, the system can fail even though none of the components do. This can happen through a design fault, which we will consider later in the chapter. The converse is also possible, that the state of the system is not erroneous even though the internal state of the system is. The internal error in the system may be masked and never affect the system environment.

It is possible for a system to accept an input where that interaction is not a continuation of a valid trace with respect to a specification P (say). The system has not failed, nor can it ever fail, with respect to P but its state is now invalid, in the sense that P can no longer be used to predict future behaviour and so to define the state of the system. However, if we have a specification which does define the behaviour of the system after such an input we can define the state of the system. The states so defined are frequently

referred to as being erroneous but we consider this to be a casual overuse of the term and we will just refer to such states as being invalid with respect to P.

5.3 Faults

Let S be a system that has failed. We now wish to consider *why* S has failed. The 'algorithm' given below does not puport to say how debugging should be carried out. It is purely a way of defining what a fault is.

Let us choose a level of detail with components C_i. Looking at the C_i we will find that the interaction, with the environment, we recognise as a failure was carried out by one of these C_i, C_n (say). Let us consider the behaviour of C_n:

1. C_n failed. The interactions which C_n took part in, until the one in question, met the specification of required behaviour for C_n and the interaction in question did not. In order to find the cause of the failure of S we now need to locate the cause of the failure of C_n. If C_n is not a basic element we choose a level of detail of C_n and carry on looking. If C_n is a basic element the failure is attributed to it.

2. C_n's behaviour did not agree with the specification of its required behaviour, but C_n did not fail. This can only happen if C_n takes part in an interaction on an import and receives an invalid input. C_n cannot be 'blamed' for the failure. We now need

to look at the source of the invalid input. Since the system has failed the source cannot be the environment. So it must have been another component C_m which produced the invalid input. So we carry on looking with that component C_m.

3. C_n has kept to its specification. Let us consider a system with two components. The first component outputs zero or one to the environment and then outputs the same value to the other component This second component then inverts the value and outputs this new value to the environment. If the first component outputs one value to the environment and a different value to the second component the system will fail when the second component produces its output even though the second component will not have failed! So when C_n has kept to its specification we need to consider all the components connected to it in order to try and find out why S failed.

So by examining C_n we have identified more components to examine. We continue to examine components until either we have a basic component which has failed or we have no more components to examine. In the first case we call the component a *fault.* In the second, since there is no failed component, there must be something intrinsically wrong with the system and we say we have a *design fault.*

It must be remembered that applying the above rules to locate a fault will give different results depending on the level of detail we begin at. The reason for this is that when we look at a component C_p (say) we might identify that it did not fail because of some

earlier invalid interaction. However, if we were to choose a level of detail where C_p was split into a number of components we might find that one of those had failed, causing the system failure, and that the invalid interaction involved another unrelated component. This problem is one of the reasons we often find it difficult to locate faults.

Consider the case where C_n has not failed but took part in an invalid interaction and the system failure can be traced to another component C_m (say). The state of C_m may be erroneous. The failure of C_m will cause some neighbouring components to move into invalid states. The components will interact with their neighbours and so move them into invalid states. After a number of these C_n will move into an invalid state. So starting from a single component containing an error larger parts of the internal state of the system become invalid. We call this *error propagation.*

When considering the case of a *design fault* we often wish to be more specific and identify what is wrong. To do this we need a 'correct' system structure to compare our 'faulty' system against. We then say that the design fault is the difference between the 'correct' and the 'faulty' system structure. In fact we can identify a 'correct' structure for the system where a component has failed and so define a fault to be the difference between the 'correct' and 'actual' structure. This is similar to our definition of error.

So a fault is that part of the system which is not as it should be. This is the cause of the error and so, by error propagation, the cause of the failure. That is, a failure is the manifestation of an error and an error is the manifestation of a fault.

5.4 Fault Prevention

The goal of fault prevention is to prevent faults from being present in the system. There are two aspect to fault prevention, *fault avoidance* and *fault removal.*

1. *Fault avoidance* is concerned with using methods and techniques which avoid the introduction of faults into the system during design and construction. For software, these techniques have made substantial progress in the last five years, particularly in the application of mathematically based methods to large problems[11]. However, there is no evidence that these methods will ever totally prevent faults being introduced into systems.

2. *Fault removal* is concerned with discovering and removing any faults which exist in the system after design and construction. Again, for software, these techniques have improved over recent years with the development of support tools and a better understanding of the underlying principles. For example, we now see static code analysis as well as dynamic testing. However, there is no evidence that we will ever have methods that will allow us to find all the faults present in a system.

5.5 Fault Tolerance

Fault tolerance is concerned with constructing systems which tolerate faults. That is, fault tolerant systems aim to provide the required behaviour even though faults are present in the system. Such an approach is necessary because, as we have noted above, systems are certain to contain faults despite the application of fault prevention techniques.

Within a fault tolerant system there are five phases of behaviour which make up the system's response to a fault: error detection, damage assessment, error recovery, fault treatment and continued service. These techniques form the basis of all fault tolerant techniques and thus should form the basis for the design and implementation of a fault tolerant system. Any approach to fault tolerance will be contain some, but not necessarily all of these phases.

5.5.1 Error detection

In order to tolerate a fault, it or its effects must be detected. Usually faults are not directly detected by the system. But, since any manifestation of the fault will generate errors, which may propagate through the system the existence of the fault can be detected indirectly by detecting an invalid state in a component. In order to be able to detect this invalid state the system must have some 'knowledge' of the correct state with which to

compare the actual state.

We must remember here that the system's external state does not neccesarily contain an error, since the measures for fault tolerance within the system may guarantee that, with the particular type of fault that exists, the system can never fail, and so the external state cannot be considered erroneous. However, it is possible that we do consider the system's external state to contain an error because the system could (later) fail but our measures manage to move the system into an error free state before this failure occurs.

5.5.2 Damage assessment

Because of the delay between the manifestation of the fault and its detection many components may now be in an invalid state due to error propagation. Before any attempt is made to make the state of these components valid, the full extent of the damage may need be assessed. The extent of this assessment will depend on decisions made by the system designers concerning damage confinement and on the techniques used for identifying damage. In fact, damage assessment may not be carried out because the designed have assumed a fixed extent for the error propogation.

5.5.3 Error recovery

A fault within a system has manifested. The state of the system, at many levels of detail, is erroneous. We have detected this and decided on which components we consider to be in invalid states. The system now needs to take some action such that the internal state of all levels of detail become error free. We call this *error recovery*. If the system does not take such action to correct its state then system failure may occur.

There are two ways in which the system can perform error recovery:

1. *Forward error recovery:* The system deduces from the current state a (hopefully) correct state. Alternatively, more than one component of the system may have carried out the computation and the system may be in a position to decide that the state of one (or more) of these components is not in error and use this 'correct' state to correct the states of the components which are in error.

2. *Backward error recovery:* The system reverts back to some stored state from earlier in the computation. It is hoped that this earlier state is correct and that the error detected was introduced later, after the time when this state was the current state.

5.5.4 Fault treatment

Although the error recovery phase may have returned the system to an error free state it is necessary, in order to enable the system to provide continued service, to ensure that the fault does not immediately recur. The techniques for fault treatment go through the same type of steps as we have outlined above for errors. This is not surprising, given the similarity between the definitions of error and fault. The fault is detected, the fault is then located and following this the fault is repaired. If the fault is thought to be temporary no action will be taken.

5.5.5 Continued service

An error is the system state is discovered, located and removed, and then the cause of this error is located and removed. The system may then continue to provide normal service. Of course, our measures for fault tolerance may not be effective and the system not be able to provide continued service.

In the following example we will ignore interactions with the environment; this is a simplification required by the lack of a connection between our model of behaviour (traces) and our model of structure (structured graphs). It does not weaken its main point.

Example 5.2 Let us consider a system S with components C and C'. In the normal

functioning of S it is required that C **sat** P and C' **sat** P'. Say C fails with respect to P by interacting illegally (according to P and P') with C'. So we have:

$$P(r,\emptyset) \wedge P'(r,\emptyset) \wedge \neg P(r \wedge \langle f \rangle, \emptyset) \wedge \neg P'(r \wedge \langle f \rangle, \emptyset)$$

Now suppose, in fact, that we have built C and C' to satisfy specifications Q and Q', respectively, and C has not failed with respect to Q. Though, because of the illegal interaction with C, C''s future behaviour is not defined with respect to P', it is defined with respect to Q'. So, C and C' now interact with each other according to Q and Q'. So we could have:

$$Q(r \wedge \langle f \rangle \wedge s \wedge t, \emptyset) \wedge Q'(r \wedge \langle f \rangle \wedge s \wedge t, \emptyset)$$

such that:

$$P(r \wedge t, \emptyset) \wedge P'(r \wedge t, \emptyset) \wedge (\forall s' \leq s : \neg P(r \wedge s', \emptyset) \wedge \neg P'(r \wedge s', \emptyset))$$

In this example we can consider s to be the error recovery and fault treatment and t to be continued service. That is, after C has taken part in s we can consider its state to have been returned to what it was after r and can now return to using P (and P') as the specification of C (and C').

We should note that this means we require additional specifications to describe the behaviour of components in the presence of failures and to describe how errors and faults are to be fixed.

Example 5.3 Figure 5.1 shows the usual triple modular redundancy pattern for a system.

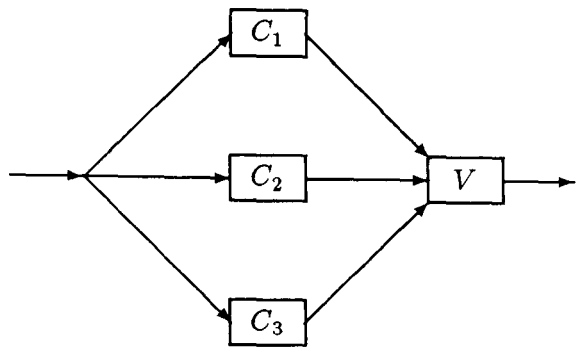


Figure 5.1: Triple Modular Redundancy

We will show how it uses the five phases of fault tolerance.

1. *Error detection:* The voting mechanism(V) detects an error in a component. The 'knowledge' of the correct state of the system comes from the assumption that "two out of three must be right". Of course, this assumption may not always be correct.

2. *Damage assessment:* It is assumed all of the state of the component which appears to have produced an incorrect output (the 'faulty' component) is erroneous.

3. *Error recovery:* There are a number of options. The 'faulty' component can be allowed to continue to function with no alteration. Alternatively, the 'faulty' component can be sent a signal indicating that it has failed and the component then attempts to correct itself. Finally, some other component can be signalled that

ensure B
by C_1
else by C_2
else error

Figure 5.2: Example of a Recovery Block

there is an error and this component takes some action to correct the state of the 'faulty' component. For example, by setting the state of the 'faulty' component from the state of another component which is believed to be functioning correctly.

4. *Fault treatment:* There are two main possibilities here. Firstly, the fault is ignored and continued service is ensured by the correctly functioning components masking the faulty component. Secondly, the system could be reconfigured to use only two components or to switch out the faulty component and make use of a new component.

5. *Continued Service:* The system continues in a (hopefully) correct state, possibly with the 'faulty' component replaced with another, hopefully, fault free component.

Example 5.4 A recovery block, figure 5.2, is another example of a fault tolerant structure, this time one used within programs. The five phases are as follows:

1. *Error detection:* C_1 is executed and then the assertion B is evaluated. If B evaluates to **true** then we assume that C_1 has functioned correctly and execution of the

program continues from the statement immediately following the recovery block. However, if B evaluates to **false** then we assume that the state is invalid. For example, if C_1 was a sort routine then B might check that the result was in the correct order.

2. *Damage assessment:* No damage assessment is carried out. It is a design assumption that the state on entry to the recovery block was not invalid and so only those parts of the state which have been modified within C_1 are damaged.

3. *Error recovery:* The state is returned to a correct state by reverting to the state before the recovery block was entered.

4. *Fault treatment:* The system attempts to avoid immediate recurrence of the fault by making use of an *alternate*, in this case C_2. However, since subsequent executions of the recovery block will again use C_1, no long term fault treatment is carried out.

5. *Continued service:* The system will either continue after the recovery block in a (hopefully) correct state, or, if the assertion cannot be satisfied by any of the alternates then, either the program will be aborted or some other measure will be needed

5.6 Faults and Errors, Programs and Programmers

Let S be a system with specification P and Q be a specification for which we are trying to construct a program which S will execute. Let p be a program with the trace s represent the loading of the program p into S. Let t be a trace and e an output interaction such that $P(s \wedge t, \emptyset)$, $Q(t, \emptyset)$, $P(s \wedge t \wedge \langle e \rangle, \emptyset)$ and $\neg Q(t \wedge \langle e \rangle, \emptyset)$. Our system has not failed with respect to P, but S **after** s has failed with respect to Q. So the failure of S cannot be the cause of the failure of S **after** s. Therefore, for the cause of the failure of S **after** s we need to look at s and, if we assume s is the correct trace to load p, then we come to p and we say that p must contain the fault.

Now, consider a program p' and let us load it into S using a trace s'. If (S **after** s') **sat** Q then p' is a 'correct' program for Q, assuming s' is the correct trace to load p'. Now p and p' will be different, these differences are *faults.*

This definition of fault is consistent with our earlier definition since when we are considering a programmed system we wish to consider the program to be part of the structure of that system. Another point to note is that if we are given S **after** s, understanding that it satisfies Q, the fault is already present in that system and so program faults are design faults

Let us now consider the introduction of faults into a program. This has been discussed

in other works in this area, for example[10]. The program, and its associated loading trace, is an output from a programming system and if the program is not correct then this programming system has failed. That is, the programming system is given as input some requirement/specification which defines what the program to be produced will do. For example, the programming system which produced p above might have been given the specifications P, the specification of the system to be programmed, and Q, the specification of the required behaviour of the programmed system. The specification of the programming system will be such that if it produces outputs, such as p and s, that mean that the programmed system does not have the required behaviour then we can say that the programming system has failed.

Let us consider the state of the programming system. We can choose a representation of the state such that we can recognise the program in the state. So, if some part of the programming system were to fail, for example a programmer makes a mistake, then it might cause, either directly or through error propagation, the program to become invalid. So looking at the external state we can see that the program contains an error, that is, it is different from what it should be. This program will become the output of the programming system, will be loaded into the programmed system, and then this programmed system may fail. So when we consider the program to be part of the state of the programming system we should refer to errors and when we consider the program to be part of the structure of the programmed system we should refer to faults. However, in general, this distinction is not important and so we will use error and fault interchangably.

Chapter 6

Conclusion

In this chapter we review the work covered by this thesis and consider what has been achieved.

We originally wished to produce a single formal model of system structure and behaviour, and to define basic concepts and terminology for fault tolerant systems in terms of that model. However, we have been unable to produce a single unified model and have instead, in this thesis, produced two separate models, one for system structure and one for system behaviour. Having separate models has not greatly limited our ability to give precise definitions of our concepts and terminology. The problems that have arisen are in the areas of state and error. Although we say "The structure we choose for our state is often based on our knowledge of the structure of the system"; since we did not relate structure

and behaviour we could not relate structure and state, and so can give no guidance on the choice of representations for external state. The inability to relate structure, behaviour and state came through in our discussion of error propogation. We were unable to discuss information flow with any real degree of formality.

However, we were able to deal with some other important concepts which are not directly relevant to fault tolerant computing, namely the two related concepts: atomicity of action and behavioral abstraction.

Structured graphs, the model of structure we have developed, are based on the idea that the relationship 'is a component of' is fundamental and can always be given meaning when we are discussing the structure of an object. This relationship gave us a tree based organisation. Other, for example 'uses', relationships between components are orthogonal to 'is a component of'. This is because other relationships which describe how we see the system to be structured only have meaning between components that are not contained within one another. We defined a 'level of detail' as a cut through the tree based on 'is a component of', and gave meaning to the orthogonal relations for any level of detail.

We developed a descriptive model of system behaviour based on the theory of traces. In the model the system interacts with its environment by the passage of objects through its interface and we considered the behaviour of the system in terms of the interactions the system will participate in and those it will refuse to participate in. We considered three different concepts of abstraction which can be applied to the interface of a system

in terms of the model.

The first is 'hiding', where we ignore the parts of the system interface which are irrelevant to the problem under consideration. For example, when considering the behaviour of our program on a time-sharing computer we ignore all other terminals.

The second is 'resorting', where we change our interpretation of the objects passing through the interface. For example, one can consider the signals passing through a terminal port on a computer as purely electrical signals, as numbers, or as characters.

The third is 'collapsing', where we aggregate parts of the interface into a single part. For example, we take 8 one-bit ports and consider them together as a single port through which passes an integer in the range -128 to +127. That the concept of collapsing is related to the concept of atomicity, which was defined using structured graphs, is obvious. What is still needed is for a formal relationship to be proven between them, but we did not attempt this.

In many ways perhaps the most suprising conclusion of this thesis is that state is an external phenomenon. That is, knowledge of the state of the system (or of a component) requires no knowledge of the structure of that system (or component). State is the changing part of a model which tells us what the system is going to do next, or more formally the minimal set of states is any set isomorphic to the set of possible future behaviours of a system (after any past behaviour). The representation for the state is

decided by the 'observer'. We have called this the 'external state' of the system. We went on to define the 'internal state' of a system at a level of detail to be the ordered set of external states of its components given by that level of detail. This view of state is, we believe, much more precise than those of previous authors, who frequently confuse the concepts of state and output. Based on this concept of state we described what we mean by programming a system and the concepts of a program and of an interpreter.

When we come to look at a system we have a lot of decisions to make concerning the 'level of abstraction' we are going to use. We need to decide upon the level of detail we are going to look at, which of the (possibly) many specifications we are going to use for the system and each of the components, how are we going to represent the external and internal states of the system, which parts of the system and component interfaces we are going to hide, resort or collapse. This is in fact the danger of the phrase 'level of abstraction'. It is hardly ever clearly defined and can be interpreted in many different ways. This is why we defined the term 'level of detail'.

We define failure as deviation of system behaviour from that defined by a specification. That is, a system fails *with respect to* a specification. We could put this loosely as a 'failure is the difference between the 'actual' and 'correct' behaviour'.

Our precision with state has allowed us to be much more precise about error. We say that the state of a system is erroneous if it is different from the 'correct' state and the system may fail at some future time, after a sequence of valid interactions with

the environment. Given our definition of state, this will require two specifications, one describing the 'correct' system behaviour and one defining the 'actual' system behaviour. Two specifications provide us with two states, since states are defined with respect to specifications. One is the 'actual' (erroneous) state, that is the state (we believe) the system is in, and one is the 'correct' state, the state the system should have been in, had it functioned correctly. We define an error as the difference between the erroneous state and the correct state. Hence, errors are defined *with respect to* two specifications.

Through consideration of how failures and errors occur we come to the concept of a fault, which we define to be the difference between the 'actual' and 'correct' structure. Because we have two separate models of behaviour and structure rather than a single unified one, we are unable to be as precise as we would have liked about the cause-effect path that leads from a fault to a failure via an error. We need a model that brings together structure and behaviour. Such a model should allow one to derive system behaviour from system structure and component behaviour, and to discuss the information flow between components.

We discussed how a program could cause a system to fail. This ought to be related more closely to work by Cristian[10]. Cristian already has formal definitions for fault, error and failure for programs. Though our defintion of a program fault is equivalent to Cristian's a formal relationship has not been shown.

We have been more precise about failure, fault and error than most previous authors,

except Cristian[10]. We have also resolved the confusion many authors have had between states and outputs. We have given a formal definition of programs and programming but this has not been related to the structure and semantics of programs.

Finally, to state our definitions in an informal way: we have defined a *failure* to be the difference between actual and correct behaviour, an *error* to the difference between actual and correct state and a *fault* to be the difference between actual and correct structure, the 'cause of a failure'.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Tom Anderson, for the help that he gave me throughout my work. I would also like to thank the many people who were at the Computing Laboratory with me who gave me help and encouragement. In particular, Dave Mundy for always having time for a chat and a never ending supply of coffee.

I would like to acknowledge the facilities (and large amounts of paper) provided by my employers Plessey Research Roke Manor Ltd. And I would like to thank my colleagues in the Software and Cognitive Engineering Division for their encouragement, particularly George Purcell for constantly reminding me that other people had made it.

Finally, I would like to thank my wife, Jackie Rippeth, for her hard work, in reviewing the thesis, and her perserverance. I wouldn't have finished without it.

References

[1] A.Avizienis. Framework for a taxonomy of fault tolerance attributes in computer systems. In *Proc. of the 10th Annual International Symposium on Computer Architectures*, 1983.

[2] T. Anderson and P. Lee. Fault tolerance terminology proposals. In *Proc. of IEEE conf. on Fault Tolerant Computing Systems, number 12 (FTCS-12)*, pages 29–33, 1982.

[3] T. Anderson and P. A. Lee. *Fault Tolerance, Principles and Practice.* Prentice-Hall International, London, England, 1981.

[4] A. Avizienis. The four-universe information system model for the study of fault tolerance. In *Proc. of IEEE conf. on Fault Tolerant Computing Systems, number 12 (FTCS-12)*, pages 6–13, 1982.

[5] A. Avizienis and J. C. Laprie. Dependable computing from concepts to design diversity. In *Proc. of IEEE*, pages 629–638, May 1982.

[6] E. Best. *Semantics, Verification and Design of Concurrent Programs using Atomic Actions.* PhD thesis, University of Newcastle upon Tyne, 1981.

[7] E. Best and B. Randell. Atomicity. *Acta Informatica*, 1981.

[8] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *JACM*, 31(7):560–599, 1984.

[9] B. Carre. *Graphs and Networks.* Oxford University Press, 1979.

[10] F. Cristian. *On Exceptions, Failures, Faults and Errors.* Technical Report, IBM Research Laboratory, San Jose, 1983.

[11] W. J. Cullyer. High integrity computing. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 1–35, Springer-Verlag, 1988.

[12] J. R. Garnsworthy. A formalisation of the concept of atomic actions. *Phillip Merlin Memorial Prize*, 1984.

[13] J. N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuler, editors, *Lecture Notes in Computer Science 60*, pages 393–481, Springer-Verlag, Berlin, 1978.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[15] J. J. Horning and B. Randell. Process structuring. *Computing Surveys*, 5(1):5–30, March 1973.

[16] S. B. Jones. *The Peformance Evaluation of Interpreter Based Computer Systems*. PhD thesis, University of Newcastle upon Tyne, 1981.

[17] H. Kopetz. The failure fault (ff) model. In *Proc. of IEEE conf. on Fault Tolerant Computing Systems, number 12 (FTCS-12)*, pages 14–17, 1982.

[18] J. Laprie. *On Impairings, Means and Measures of Computing Systems Dependability: Basic Concepts and Taxonomy.* Technical Report 83.050, Laboratoire d'Automatique et d'Analyse des Systems, July 1983.

[19] J. Laprie and A. Costes. Dependability: a unifying concept for reliable computing. In *Proc. of IEEE conf. on Fault Tolerant Computing Systems, number 12 (FTCS-12)*, pages 18–21, 1982.

[20] D. B. Lomet. Process structuring, synchronization and recovery using atomic actions. *SIGPLAN Notices*, 12(3):128–137, March 1977.

[21] P. M. Melliar-Smith and B. Randell. Software reliability: the role of programmed exeception handling. *SIGPLAN Notices*, 12(3):95–100, March 1977.

[22] D. L. Parnas. On a buzzword: hierarchical structure. In *IFIP Congress 74*, pages 336–339, August 1974.

[23] A. Robinson. A user oriented perspective of fault tolerant system models and terminologies. In *Proc. of IEEE conf. on Fault Tolerant Computing Systems, number 12 (FTCS-12)*, pages 22–28, 1982.

[24] S. K. Shrivastava. On the treatment of orphans in a distributed system. In *Proc. Third Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, IEEE Computing Society, 1983.

Notation

This appendix lists the notation used in the thesis:

$\emptyset$	Empty set	
$\{,\ldots\},\{:\}$	Set constructors	$\{a,b,c\}$, $\{x : P(x)\}$
$\cup$	Union	$\{a,b\} \cup \{b,c\} = \{a,b,c\}$
$\cap$	Intersection	$\{a,b\} \cap \{b,c\} = \{b\}$
$\in$	Element of	$a \in \{a,b,c\}$
$\subseteq$	Subset	$\{a\} \subseteq \{a,b,c\}$
$\setminus$	Set difference	$\{a,b\} \setminus \{b,c\} = \{a\}$
$X \times Y$	Powerset	$\{a\} \times \{b,c\} = \{\{a,b\},\{a,c\}\}$
$\wedge$	Logical and	$a \wedge b$
$=$	Equal	$a = b$
$\Rightarrow$	Implications	$P \Rightarrow Q$
$\exists$	There exists	$\exists x : P(x)$

$\forall$	For all	$\forall x : P(x)$
$\hat{\Gamma}^{+}(x)$	Accessible, see page 35	
$\hat{\Gamma}^{-}(x)$	Converse-accessible	
$\hat{\Gamma}(x)$	Neighbours	
$\overset{*}{\Gamma}^{+}(x)$	Accessible, including self	
$\overset{*}{\Gamma}^{-}(x)$	Converse-accessible, including self	
$\overset{*}{\hat{\Gamma}}(x)$	Neighbours, including self	
C^{+}	Descendants of a set of nodes	
C^{-}	Ascendents of a set of nodes	
$\hat{x}_T$	Leaves of a subtree of T, rooted at a node x, see page 42	
$G[A]$	Collapsing of a node A in a level of detail G, see page 45	
$[A]G$	Opening of a node A in a level of detail G, see page 46	
$\prec\!\!\cdot$	Immediately precedes, see page 50	$e \prec\!\!\cdot\, e'$
$\prec$	Precedes $e \prec e'$	
$\parallel$	Concurrent with $e \parallel e'$	
.	Pair constructor (c.f. Lisp)	$m.x$
$\langle\rangle$	Empty list	
$\langle,\ldots\rangle$	List constructor	$\langle 1,2,3\rangle$
$\wedge$	List append	$\langle 1\rangle \wedge \langle 2,3\rangle = \langle 1,2,3\rangle$
*	Closure	$\{1,2\}^{*} = \{\langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 1,2\rangle\}$
$\star$	Map function over list	$\mathrm{inc} \star \langle 1,2\rangle = \langle 2,3\rangle$

$\triangleright$	Filter list, removing those that match	$\text{even} \triangleright \langle 1,2 \rangle = \langle 1 \rangle$
α	Alphabet, see page 63	
i, **o**, **s**	Inports, outports and sorts, see page 61	
after	After, see page 67	
ren	Rename, see page 68	
hide	Hide, see page 71	
res	Resort, see page 75	
col	Collapsing, see page 84	