A formal basis for some dependability notions

Cliff B Jones

School of Computing Science,
University of Newcastle
NE1 7RU, UK
e-mail: cliff.jones@ncl.ac.uk

Abstract. This paper shows how formal methods ideas can be used to clarify basic notions used in the field of dependability. Central to this endeavour is fixing a notion of system. Relationships between systems are also considered: in particular, the importance of the situation where one system is generated by another (possibly human) system is explored. The formalisation is used as a basis for definitions of the notions of *fault*, *error* and *failure*. Some applications to examples from the dependability literature and extensions of the basic model of system are also sketched.

1 Introduction

This paper is written by a researcher from the "formal methods" community who has been trying to understand notions used in work on "Dependability". The term *dependability* is used broadly to cover many aspects such as reliability, integrity, correctness (with respect to a specification), availability, security and privacy. To begin to understand such a range of concepts, this paper starts at the bottom: the notions of *fault, error, failure* in [Lap92,Ran00]) appear to be both useful and widely accepted. The idea here is to offer definitions of these terms with respect to a particular notion of what constitutes a *system.* This choice is discussed in Section 2 and an initial formalisation is outlined in Section 3. More realistic notions of system are discussed in Section 5.

The intention here is not to offer formalism for its own sake. In fact, the details of the particular notation etc. are unimportant. The hope is that understanding can be increased by building on a firm foundation. The first fruits of the formalisation are given in Section 4 where, among other things, some relationships between systems are explored: the propagation of *fault, error, failure* chains where one system is *built on* another system is well understood; many of the failure propagation situations of interest in socio-technical systems arise where one system is *created by* another. Lastly, the idea of one system being *deployed with* another system is considered.

A concluding section lists some topics for further work including how the system notion can be enriched.

2 Systems

There are many, rather different, things that can be loosely viewed as "systems". Few would decline to use the term for an airline seat reservation system. In such a system it is easy to see an interface and the effect of using this interface. For example, requests can be made and information displayed that reflects the contents of a "database"; other interactions change the database of reservations so that (as well as reserving a seat for a passenger) similar requests for information at two points in time give different results. Other computer systems are linked to processes that evolve autonomously. In, say, an automatic braking system (ABS) for a car, the speed of rotation of a wheel is influenced by the friction of the road surface which is not under control of the ABS.[1] Less conventionally, the "system" notion here includes those where humans play an essential role. The adjective "socio-technical" is often used for such systems. The term *computer-based system* is used here to indicate that, while a computer is involved, a study of the system which ignores the human involvement is hopelessly inadequate.

[1] Such systems are called "open" in [Kop02a] in contrast to a "closed" system where all essential interactions are under control of the program. The target formalisation must cover both closed and open systems. See [Kop02b] for more on Real-Time Systems.

The emphasis in the *Interdisciplinary Research Collaboration* (IRC) on the *Dependability of Computer-Based Systems*[2] is on such systems. Although taken for granted here, it is not difficult to argue that a computer-based system can only be made dependable by examining the roles of the people involved. This should not be equated to an attempt to view humans as machines. It can, in fact, be argued that understanding potential human failure is a key step towards designing systems that are safe and pleasant to operate or use.

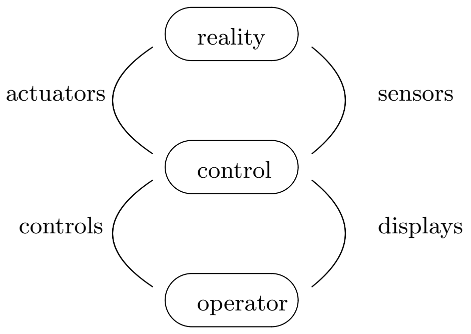

Fig. 1. A controlled system

To indicate the range of things that need to be covered, consider motor cars. At a coarse level, one could distinguish the mechanical car from the human operator or driver. In a modern car, however, the mechanics of the brakes etc. are controlled by computer systems with which the operator actually interacts. A typical linking of three systems (operator, control and (relevant portion of) reality) is pictured in Figure 1.[3] Each of these three classes of system has distinct properties. An operator is likely to make unpredictable errors[4] whose frequency will vary over time; the likelihood of such errors will be influenced by how easy it is to understand the interface of the control system. The control system receives signals from actuators operated by the human and from sensors that are intended to measure aspects of reality. To earn the name "control system", it would have some degree of autonomy in sending signals to the actuators; it would also provide feedback to the operator. The reality is a system itself which will evolve even without inputs: for example, a car will continue to move while the driver tries to decelerate. Each of the components indicated in Figure 1 can be viewed as a system. The notion is also recursive in that the whole can be usefully

[2] *DIRC*: Consult `www.dirc.org.uk` for information about the Dependability IRC.
[3] The additional concept of an *advisory system*, together with an analysis of its failure modes, is studied in [SO02] .
[4] See [Rea90] for a classification of human errors.

viewed as one system for some purposes; it is also easy to see that each of the systems might be further decomposed.

Because there can be several interlinked systems to consider, there is always a danger of confusion over *system boundaries*. A lack of exactness here can give rise to pointless debates about the cause of a problem with a system. The topic of *system boundaries* is picked up in Section 4.2.

For many systems, it is useful to discuss a distinction between their externally observable behaviour (a "black box" view) and their internal design. For example, a "transparent box" view of computer software involves looking at its code. Many interesting properties of a system can be discussed via its "black box" view that concerns *only* the behaviour at its interface. To make these issues precise, it is time to turn to the topic of formalisation.

3 Formalisation

This section initiates the formalisation effort by considering simple systems; this is not the limit of ambition of the paper but even such simple systems suffice to illustrate some benefits of formalisation. The notion of system is enriched in Sections 4.3 and 5. What follows is not claimed to be deep: it is a selection of known concepts from formal development methods. The intention is to show that these ideas make it possible to discuss core dependability notions more precisely than in the absence of a formal foundation.

3.1 Simple systems

To view something as a system, it is necessary to choose an interface. An interface offers a collection of ways of interacting with a system: here, interaction points are termed "operations". So, for example, a seat reservation system is likely to offer some enquiry operations and others for reserving places. One requirement, then, for pinning down an interface to a system is to list its operations. For a simple (computer) system, each operation is likely to have a name. In addition to the name of each operation, it is likely that there will be information that is either required as input to narrow down the operation (parameters to an enquiry might be starting and finishing locations) or information which is returned as a result of the operation.[5] The parameters and results of software operations are normally recorded by some form of type annotation but, rather than worrying about a specific notation for this, there is a more fundamental concept to be understood.

If systems with even simple databases are to be described, it is necessary to find a way of discussing how operations are affected by –and in turn affect– that database. The notion of *state* can be used to generalise situations from those where a system updates a simple counter to those where a large database is

[5] One could discuss whether there is a single "reserve" operation or a whole set thereof; for practical purposes it is worth grouping like operations together even though this necessitates separating the idea of parameters and results.

required. Thus the formalisation of the behaviour of an operation will relate, not only its parameters and results, but also state values. Is it true that all systems, have a state? Strictly, no! A pure function like square root could well deliver the same result (for a given parameter) at any time; even a sorting system might behave like a mathematical function whose result does not depend in any way on the history of its use. Be that as it may, it is useful to say that systems do have states and to regard functions as a special case.

So a system will be characterised by its states and its set of operations. To go further, formulae will be used — but remember what is made clear in the introduction: these formulae indicate a possible notation; they are not the main point of the paper.[6]

A description[7] of a system comprises a class of states (Σ) and a set of named operations. The states used in a specification will typically be an abstraction of those of any particular implementation. The set of states can be defined, for example, as VDM objects with the possible use of a *data type invariant* which records that subset of the overall set of objects that can arise.

In simple cases, each operation OP_i would be a function from states to states $OP_i\colon \Sigma \to \Sigma$. More generally, operations are not defined for all possible starting states (or combinations of states and parameters). The fact that operations can be *partial* is recorded in VDM or B by noting a *pre-condition* that defines, what might be called, the "termination set" of an operation. (The notion of a pre-condition is a simple example of an assumption about the deployment of a system; this is key to the discussion in Section 4.4.)

The topic on *non-determinism* is important even for systems which will actually behave deterministically (i.e. always deliver the same final state and result for a given starting state and parameters). It is often convenient in a specification or description of a system to leave open the possibility of different results. In order to record this, the effect of an operation is given by a relation over states.

Thus, the termination set for OP_i is a set of states $T_i\colon \mathcal{P}\Sigma$ and the result is a relation over states $M_i\colon \mathcal{P}(\Sigma \times \Sigma)$. It is required for $Op_i = (T_i, M_i)$ that the meaning relation gives possible results for any state in the termination set $T_i \subseteq \mathbf{dom}\, M_i$.

To make this more concrete, consider the following description of a *Stack* of natural numbers ($\mathbb{N}$). A suitable model of the state is a sequence of those numbers that have been *Push*ed onto the *Stack* but not *Pop*ped off: $\Sigma = \mathbb{N}^*$. A plausible list of operations might be $Ops = \{Init, Push(i), Pop, Top, IsEmpty\}$. The *Pop* operation ($Pop = (T_{Pop}, M_{Pop})$) only works (is defined) for non-empty states and removes the top element of a stack (but returns no result)

$$T_{Pop} = \{l \in \mathbb{N}^* \mid \mathbf{len}\, l \neq 0\}$$

[6] The notation used here is motivated by VDM (cf. [Jon90]) but avoids specific details and could trivially be translated into other notations such as Z (cf. [Hay93]) or B (cf. [Abr96]).

[7] The term *specification* indicates some official status; it is often more useful to discuss "descriptions" of systems. There can be different descriptions for different purposes.

To emphasise that the meaning can be stated as a relation, it is written as

$$M_{Pop} = \{([e] \frown l, l) \mid \ldots\}$$

Notice that $T_{Pop} \subseteq \mathbf{dom}\, M_{Pop}$ but that the meaning relation would not be defined for empty starting states.

In this case, the result is deterministic but the same style could be used to specify an operation that yielded an arbitrary element of the stack

$$M_{Arb} = \{(l, l(i)) \mid 1 \leq i \leq \mathbf{len\, l}\}$$

3.2 Concepts from development methods

There are a number of established development methods (e.g. VDM [Jon90] or B [Abr96]) that are built around state-based specifications and which contain useful, thought out, concepts that can be utilised when discussing systems.

The cornerstone of any formal development method is a precise notion of a specification. The desirability of separating a specification from any implementation is one of the tenets of this paper. It could be extremely wasteful if users were forced to understand the detail of an implementation in order to use an artifact. Moreover, denying users a specification is likely to result in them detecting properties that the developer regards as accidental; but any subsequent change of these "irrelevant details" might offend the users. A classic example of this was seen when early computer architectures were simulated on newer, faster, machines: users frequently found that their legacy programs no longer worked because they had relied on undocumented features of the instruction set. Crucially, it is difficult to see how an engineering process can be used to create a system where there is no initial notion of specifying the required properties of the to-be-created artifact. The concept of what is required might evolve (see Section 5) but at any time there ought be something other than the implementation which defines the functionality.

There are different styles of specifications and there was at one time a claimed dichotomy between model-oriented and property-oriented specifications but this is now largely resolved by recognising that *property-oriented* specifications are more conveniently used for basic data types whereas *model-oriented* specifications are better for systems with more complex states.

A *model-oriented* specification (as in VDM, Z or B) is built around a *state* that is an abstraction of internal values retained by a system between uses. The *Stack* example above retains the elements that have been *Push*ed onto (but not yet *Pop*ped from) the stack; many operating systems store user preferences (and maybe desktops) between invocations; or a database system might contain enormous amounts of data about an enterprise. It is important to remember that a good specification will be built around an *abstract* state and the VDM literature (e.g. [Jon90, pp 216–219]) includes a discussion of specification "bias" and offers a precise test for sufficiently abstract states.[8] For now, it is only

[8] An unbiased state is one whose equality can be tested by the operations of a system.

necessary to accept that the idea of using an abstract state in a specification does not constrain implementations. What is specified is the behaviour as observed via the operations of a system; the state is a convenience for the description.[9]

Methods like VDM offer a precise notion of what it means for a system to *satisfy* a specification. In fact, this question can be addressed at the level of specifications. To say that system B satisfies the specification of system A, is to claim that for each operation of A, there is one in B that respects the specification in A. An operation b respects a if it terminates for at least the required termination set of a and is more deterministic than a (over the termination set of a). Formally OP_i **sat** OP_s where $OP = (T, M)$ iff $T_s \subseteq T_i$ and $T_s \lhd M_i \subseteq M_s$. There are important properties of this relation: it is, for example, reflexive and transitive; furthermore, a reasonable set of programming constructs are monotone in the relation. These mathematically expressed properties make for a useful development method.[10]

There are two approaches to such proofs. The most commonly used *data reification* proof obligation in VDM relies on finding a *retrieve function* (homomorphism) from the state of the implementation to that of the specification. This rule suffices for most steps of development but is based on the assumption that development always proceeds by adding detail. Lynn Marshall [Mar86] identified an important case where this was not true and a complete rule that uses a relation between the specification and representation states is described in [Nip86,Nip87].

The topic of concurrency has proved challenging for formalisation. Whereas the appropriate form of specification for sequential systems is uncontroversial, pinning down a suitable specification notion for concurrent systems has led to many experiments. The key challenge (see [Jon00,dR01]) has been to support *compositional* development of concurrent systems. One approach is the use of rely and guarantee-predicates to record assumptions and limits of *interference* for an operation. Although [Jon83,Stø90,Col94] etc. apply the idea to shared-variable concurrent systems, it is not difficult to argue that the notion of interference is at the heart of what is meant by concurrency and that taming it is also essential in communication-based concurrency such as embodied in process algebras. (Stirling describes a related approach in [Sti88].) While the details of such ap-

[9] Because of the widespread literature on property-oriented specifications, it is worth pausing to repeat comparisons that are elaborated elsewhere (e.g. [Jon89,Jon99]). In the 1980s, model-oriented specification methods were much less written about than so-called "Algebraic Specifications" of data types [AKKB99]. The objective of an *algebraic specification* is to fix the meaning of the operations (or functions) solely in terms of their interrelationships. For example, in an algebraic description of a *Stack*, a key equation is $pop(push(i, s)) = s$. It is actually interesting that such descriptions can ever fix a system and it does work for small systems like a simple stack. In some cases, the implied model (technically, equivalence classes of the word algebra of the functions) is by no means minimal and even –for example– a definition of a *Queue* is far less obvious than that of a *Stack*.

[10] A discussion of "compositionality", its importance, and why it is much harder to achieve for concurrent programs can be found in [Jon00].

proaches to concurrency are not germane here, it is important for Section 4.4 to observe that concurrent operations can also be described formally including assumptions on the environment in which the operations will be deployed.

For what follows, the single most important point from this section is that there is a precise notion of satisfying a specification.

4 Some uses of the formalisation

This section begins to explore what can do done with the formalisation of even simple systems.

4.1 Dependability notions

The division of what might loosely be called "problems" into faults, errors and failures in [Lap92,Ran00] is a significant contribution to our ability to discuss causes and effects. This section re-examines this basic terminology with reference to the notions in Section 3.

The term *failure* in [Lap92,Ran00] is described as "A system *failure* occurs when the delivered service deviates from fulfilling the system *function*, the latter being what the system is *aimed at.*" (all emphasis in the original). Presumably the intention of the phrase "what the system is aimed at" is to acknowledge that a specification could itself be wrong or even not exist. The position taken here is that an implementation can only be judged to fail if there is a specification against which a deviation can be identified. Thus, a failure of an operation is a behaviour that is not in accordance with the specification of that operation. (Questions of missing or erroneous specifications are addressed in Section 4.3.) An appropriate definition might therefore be

> A system *failure* (with respect to a specification) occurs when the delivered service deviates from that specification.

Brian Randell in [Ran00] has made the additional point that failure is a *judgement* that is made by another system. This observation sits nicely with the view here. Clearly, one system can observe the behaviour of another and emit messages that claim to report the detection of erroneous results. But such *Beckmesser*-like systems are potentially just adding confusion unless their complaints are based on a specification of the required behaviour. As the allusion to *Die Meistersinger* indicates, the judging system could well be human. It could also be wrong (in its judgements)! It is illuminating to consider how the notion of one system judging another ties in with law courts. There is a pleasing recursiveness in the way that a higher court can sit in judgement on the decisions of a lower court and that judgement is in turn subject to the review of a yet higher court. It is also known for non-legal institutions (history or –more worryingly– the press) to express opinions on the wisdom of judgements from even the highest courts. Courts do, of course, make their judgements against a codified specification of behaviour: the law. They rely on evidence of deviations from the law. It is thus possible to have a system in which error reports from some sub-systems are "ignored" or overridden in a larger system.

To return to the domain of computer (based) systems, the main point being made here is that the notion of a failure can only be judged against a specification and such a judgement can only be made by another system. As indicated in Section 3.2, a specification is anyway required in order for any plausible development process to deliver a product that has a chance of satisfying its specification.[11]

The description of *errors* in [Ran00] is worded in terms of a state ("An *error* is that part of the system state which is *liable* to lead to subsequent failure; an error affecting the service is an indication that a failure occurs or has occurred.") which fits well with the state and system notions of Section 3. It is quite reasonable to say that an underlying cause (fault) can give rise to erroneous values in a state (the question of whether this is the abstract state is considered below). The qualification "liable to lead to subsequent failure" can be taken to indicate that erroneous values in a state might never be detected. This situation can arise if particular sequences of operations are avoided (e.g. recording a wrong salary in a database might not be detected if the employee leaves the organisation). Erroneous values in a state *can* result in failure – but whether or not they do depends on the operations used.

One might question whether the wording "that part of the system state" precisely accords with the definition given here but this question can only be judged by the original authors. The definition suggested here is

> An *error* is present when the state of a system is inconsistent with that required by the specification.

There are actually two sorts of inconsistencies to be considered. The implementation state is likely to be a reification of that in the specification (a set might be represented by, say, a doubly-linked list). If the retrieved state is not that required by a specification, then subsequent operations can expose a wrong result. When this happens an error has occurred and a subsequent failure may follow. But it is also possible that the representation state is internally inconsistent (e.g. forward and backward list pointers do not agree): in this case an implementation can malfunction in ways which are not easy to predict from the specification. There is here a subtle question about the level of abstraction of the notion of state in the definition of "error". It is certainly desirable to be able to describe behaviour in terms of abstractions of particular state representations chosen in implementations. Unfortunately faults can manifest themselves at a low level of abstraction. This is a serious obstacle and is one that is likely to need some layering idea in the implementation to contain errors of this kind.

It is a corollary of the definition of fault given here that

[11] Thus far, only logical failure of a single invocation of an operation has been considered; in order to discuss stochastic properties it is obviously necessary to define requirements on (or measure) multiple observations.

Whether or not a failure occurs depends on operations invoked subsequently

The notion that something causes the erroneous state which has been taken to be the internal manifestation of an error is useful. But the notion of *fault* is described in [Ran00], "The *adjudged or hypothesised cause* of an error is a *fault*." Invoking judgement here could be argued to confound a cause and the process of determining that cause. It might be preferable to be more definite about the link with errors — here, the definition

The cause of an error is a *fault*

is suggested. It is of course true that attributing the cause is a judgement (which can be erroneous). But as with failure, this is the task of a separate system. It is interesting to observe that any judgement of what fault caused an error is likely to need access to the internals ("transparent box" view) of a system.

In the definitions offered here, there is a clear separation between what might be thought of as absolute truth and the judgements which are made about failures and their causes. The notion of probability has also been avoided so far by focusing on single failures; the separation of logical and stochastic properties is maintained until some other issues are clear.

4.2 System boundaries

A source of significant confusion in trying to attribute faults/errors/failures is the question of which system is being discussed at any point in time. Behind the whole of this section so far is the assumption that the boundary of a system is agreed. While this will not always be the case, it must be clear that little consensus or understanding can follow from a discussion where the boundary of a system is itself a matter of dispute. For example, a discussion of a rail disaster between someone who observes the effect of a broken rail on a high speed train and someone who is claiming that long term financing is the cause of all ills is unlikely to be edifying. An agreement on the nesting of systems might help the discussants understand each others' viewpoint.

It is not always true that the boundary of a system is a clear and static concept. Even before coming to socio-technical systems below, there are difficulties to be considered. The topic of multiple interfaces is considered in Section 5. It is also true that interfaces can change over time: on the micro-level, the topic of *rendez-vous* is also considered in Section 5. But the question of interfaces which actually evolve is a whole avenue of research which is recently receiving attention (e.g. [JRW02]).

4.3 Systems that create systems

A significant contribution of the separation of faults/errors/failures is that the terms can be used to explain causal chains with the failure of one system giving rise to a fault in another. This is easy to see in the case where system B is *built on* system A. An example is where software runs on some hardware in which an (uncorrected) hardware memory problem gives rise to a failure of the hardware to meet its specification; this is seen as a fault for the software (that is likely to result in an erroneous value in the state of the running program). The purpose of this section is to explore another relationship between systems.

The fact that a failure in one system can cause a fault in another is particularly interesting in the case where one system *creates* another. A simple case of this is a manufacturing system that creates products: many sorts of failure in manufacturing can result in faulty products. A specific example with computer software is that a failure in a compiler can introduce faults into a program which would have otherwise fulfilled its specification.

Thus a failure of the creating system A can give rise to a fault in the created system B. The interest in DIRC extends to systems in which humans play a role both by interacting with a final system and by their involvement in creating such a system. In the former case, the concern might be with the way a system embeds in a group of people; in the latter case, the consequences of the fact that humans design or implement a system are considered.

One can view –as a system– the organisation that creates another system. Thus the development organisation which designs and develops a program can be thought of as a system that creates programs. These programs can, in turn, be seen as systems which create computations.[12]

In an example like the Therac-25 [LT93], it is clear that failures in the development organisation created a system that contained potentially fatal faults. The deployed system included a faulty program and physical apparatus without a secondary hardware guard. One can go further and analyse faults in the development process and how they might have been caused by management failures but that is not the objective here.

Good software development processes use some form of review or inspection. This is a place where the program text (i.e. the "transparent box" view) is necessary; this would also be true if one were trying to make the judgement –without testing– whether a program was "fit for purpose". This is a form of fault-tolerance applied within the development process. As in most human processes, inspections etc. apply some form of (human) diversity to increase the chance of catching errors.

It is now easy to deal with the issue –alluded to in Section 4.1– of missing or erroneous specifications. The position taken above is that the judgement that a system fails can only be made against a specification. What if the "specifica-

[12] It is worth saying a word in defence of the much maligned programmer: there is an inherent difficulty in writing a program which will be mindlessly executed by a computer. When constructing instructions for humans, there is the chance that a stupid instruction will be questioned. One only need look at the chaos created by the "work to rule" form of industrial action to see that humans rarely follow rules as mindlessly as computers. Computers derive their power and their weakness from the fact that they always work to rule!

tion is wrong"? Presumably, this means that the specification is in some sense inappropriate; the specification might be precise, but it can be seen to result in faults and failures in a bigger system. For example, a specification might state that a developer can assume that the user will respond in one micro-second — but failing to so do can result is fatal consequences. The developer writes a program which "times out" after one micro-second and an accident occurs. It is surely not right to say that the software system (which meets its specification) is failing. Nor, of course, is it reasonable to blame "operator error" with such an unreasonable assumption. The only reasonable conclusion is that it is an earlier system which exhibited erroneous behaviour: the act of producing the silly specification itself is the failure that causes a fault in the combined system of software and operator. The judgement that a specification is "silly" must of course be made by another (external) system. A similar argument can be made for missing specifications: an engineering process requires a reference point.

There is one more relationship between systems that is interesting to consider: the idea of *deploying* one system with another has already been hinted at and is discussed in more detail in the next section.

4.4 Deploying systems

A system A can be deployed with another B[13] and they may, or may not, live happily ever after. Even with the simplest of systems discussed above, operations were specified with pre-conditions. For an artificially created system, the assumptions (pre-conditions) are an invitation to the developer to ignore certain possibilities. It is clear that robust systems should avoid unnecessary assumptions but almost no useful system can be created without some assumptions. It is also this author's experience that specifiers are much more likely to overlook assumptions (pre or rely-conditions) than to fail to describe the intended function.

It is a mistake to deploy two systems which do not respect each others assumptions. Thus, a deployment of the *Stack* in a situation where more *Pop* than *Push* operations can be used is a faulty deployment and is likely to result in a failure of the combined system.

Section 3.2 refers to work on concurrency that enriches the notion of assumptions and commitments by using rely and guarantee predicates. These can be used to discuss the appropriate deployment of systems whose behaviour is influenced by concurrent processes.

Nothing in the idea of recording assumptions makes, in itself, deployment safer. It is like recording warnings on the box of an electrical product: if the warnings are ignored, the customer might still be electrocuted. But the recording of assumptions does, like the commitment part of a specification, at least make it possible for the deployment process to be undertaken circumspectly.

[13] It is tempting to think of this relationship between systems as embedding one system in another; here a more symmetrical view like a cross-product is preferred.

The discussion in this section also links to hazard analysis in that there is now a prompt to distinguish between hazards which can result from components failing to satisfy their specifications and deployment errors where (assumptions of) specifications do not match.

4.5 Error recovery

There is an extensive literature on “fault tolerance” and “error recovery”. Given the definitions chosen in Section 4.1, it should be clear why the latter term is preferred here. Space does not permit a full analysis of concepts like forward and backward error recovery or exception handling here but some simple points are worth making.

It is well understood that simple replication redundancy (as in “Triple Modular Redundancy”) can be used to guard against random faults or decay. Software can deploy *redundancy* in a way which prevents single faults coming from data corruptions resulting in failures with respect to the specification of the software system. It should also be clear that some form of diversity is required to make any impact on guarding against design failure.[14] Given that the scope of DIRC is computer-*based* systems, the ambition is to be able to describe tolerance of human failures. In this area, the distinctions in [Rea90] between “slips”, “rule-based mistakes” and “knowledge-based mistakes” is interesting. It might be possible to record assumptions about certain classes of operator error.

One more topic which is worth addressing is the view that there are often “multiple causes” of a failure. This is commonly discussed in accident post-mortems. The point made in Reason’s graphic “Swiss Cheese” picture in [Rea97] is that any layer of fault-tolerance will have residual holes (thus the slice of cheese) and that an accident occurs when circumstances are such that the holes coincide. The position here is that *each* of the nested systems has failed. It might also be true that (as in the Therac-25 case) the human system which created the physical system deployed too few layers of protection or that the layers were insufficiently diverse.

4.6 Further stimuli

It would be profitable to reconsider many of the points made in [Per99] in the light of the formalisation above. To cite just one example, designing the architecture of a system so that user can both understand and predict its behaviour is key to the dependability of computer-based systems.

A related source is the recent research of John Rushby on “pilot errors”. Rushy [Rus99] uses finite state diagrams to describe systems. The notion of system description in Section 3 of this paper is more general but would not serve Rushby’s aim of automatic analysis. More controversially, Rushby requires a finite state description of the pilot’s perception of the control system (in order

[14] There is evidence (a joint paper with Ian Hayes and Michael Jackson is making slow progress) that forms of rely-conditions can be used to describe fault tolerance.

to perform state exploration and locate inconsistencies). The difficulty of getting users to couch their understanding of a system in this way is conceded in [Rus99]. It would be interesting to explore other approaches (such as rely-conditions) and to question whether the user's view is actually of the control system or of the reality which is ultimately being controlled (see Figure 1).

5 Further work

This paper makes only a beginning and clearly much more needs to be done with a cooperation between formalists and dependability researchers. Some of these topics require further research but the direction of the work to be done is relatively clear.

The notion of system in Section 3.1 is deliberately simple and there are many ways in which enrichment is desirable. Principal among these is for systems that interface with the physical world where states evolve autonomously.

Many systems deployed in safety-critical applications involve sensors which link to physical phenomena like temperatures. This is of course the distinction referred to above "closed versus open" systems (cf. [Kop02a]). One problem to be faced here is that values are time-indexed quantities. This issue has been faced by several researchers (e.g. Duration calculus, Mahony/Hayes[MH91]).

It is also imperative to recognise that systems themselves evolve. In the DSoS project (Dependable Systems of Systems), building systems from components which are not under the control of a central organisation is a major concern. The interfaces of such components can change without consultation. Rather than just say that such systems will never work, the approach being researched in DSoS is to understand to what extent such interface evolution can be brought under the control of an exception handling view (see [JRW02]).

Another issue which is being considered in the DSoS project is the fact that it is frequently convenient to view a system as having multiple interfaces. Hermann Kopetz talks of service, diagnostic and configuration interfaces. This idea might usefully be generalised because something like evolving the topology of a system is likely to have different behaviour than the basic operation interface. But it does not appear that any new conceptual problems are introduced by splitting the view of a system's interface in this way.

The simple model of Section 3.1 assumes that any operation is executed atomically (without interruption) and also that all operations are available at any time. It is often argued that this is a reason for rejecting this model and moving to a process algebra which can express the fact that the changes in which operations are available can be expressed. In fact, it is not difficult to express interfaces with such a *rendez-vous* behaviour. Several object-oriented languages such as POOL [Ame89] offer, in addition to the methods of a class, a process per class which executes in each object of that class to say which methods are "available" at any time.

There is also the question of the meaning of the notation for describing systems. If development methods are to be based on particular notations for describing systems, there must be a firm semantic foundation in terms of which the correctness of results like the monotonicity of refinement can be argued. For VDM, (sets and) relations suffice (see [Jon87]) to prove, for example, that the satisfaction relation is transitive and that it is monotone in standard program combinators. For the sort of concurrent object-based language mentioned above, the π-calculus (see [MPW92,SW01]) has proved ideal (see [Wal91,Jon94,San99]). For systems that create systems, some form of higher-order π-calculus might be required.

Acknowledgements

This paper owes its inception to the EU-funded project known as "DSoS" (IST-1999-11585) and the author is grateful to colleagues on that project for many useful discussions and patient hearings of early versions of the ideas presented here. The work has also been supported by the EPSRC-funded "Interdisciplinary Research Collaboration" on "Dependability of Computer-Based Systems". Here again members of that project have made many useful comments. The author would like to thank in particular Denis Besnard, Sadie Creese, John Dobson, Ian Hayes, Michael Jackson, Michael Jones, Tony Lawrie, Nick Moffat, Brian Randell, Carles Sala-Oliveras, Ian Sommerville and an anonymous referee for their sustained interest in –and stimulating criticisms of– drafts of the material.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[AKKB99] Egidio Astesiano, Hans-Jorg Kreowski, and Bernd Krieg-Bruckner, editors. *Algebraic Foundations of Systems Specification.* Springer-Verlag, 1999.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY.* PhD thesis, Louvain-la-Neuve, June 1994.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Cambridge University Press, 2001.

[Hay93] Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International, second edition, 1993.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon87] C. B. Jones. Program specification and verification in VDM. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F: Computer and Systems Sciences*, pages 149–184. Springer-Verlag, 1987.

[Jon89] C. B. Jones. Data reification. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 79–89. Butterworths, 1989.

[Jon90] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In A. W. Roscoe, editor, *A Classical Mind*, chapter 14, pages 231–246. Prentice-Hall, 1994.

[Jon99] C. B. Jones. Scientific decisions which characterise VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.

[Jon00] C. B. Jones. Compositionality, interference and concurrency. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Milennial Perspectives in Computer Science*, pages 175–186. Macmillian Press, 2000.

[JRW02] Cliff Jones, Alexander Romanovsky, and Ian Welch. A structured approach to handling on-line interface upgrades. In *(to appear in) proceedings of COMPSAC*, 2002.

[Kop02a] Hermann Kopetz. On the specification of linking interfaces in distributed real-time systems. Technical Report 2002/8, Institut fuer Technische Informatik, TU Vienna, 2002.

[Kop02b] Hermann Kopetz. *Real-Time Systems*. Kluwer, 2002.

[Lap92] Jean-Claude Laprie. *Dependability: basic concepts and terminology—in English, French, German, Italian and Japanese*. Springer-Verlag, 1992.

[LT93] N. G. Levenson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, pages 18–41, July 1993.

[Mar86] L.S. Marshall. *A Formal Description Method for User Interfaces*. PhD thesis, University of Manchester, 1986.

[MH91] B Mahony and I Hayes. Using continuous real functions to model timed histories. In P. Bailes, editor, *Engineering Safe Software*, pages 257–270. Australian Computer Society, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Nip86] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[Nip87] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types*. PhD thesis, University of Manchester, May 1987.

[Per99] Charles Perrow. *Normal Accidents*. Princeton University Press, 1999.

[Ran00] B. Randell. Facing up to faults. *The Computer Journal*, 43(2):95–106, 2000.

[Rea90] James Reason. *Human Error*. Cambridge University Press, 1990.

[Rea97] James Reason. *Managing the Risks of Organisational Accidents*. Ashgate Publishing Limited, 1997.

[Rus99] John Rushby. Using model checking to help discover mode confusions and other automation surprises. In *Proceedings of 3rd Workshop on Human Error*, pages 1–18. HESSD'99, 1999.

[San99] Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

[SO02] Carles Sala-Oliveras. Systems, advisory systems and safety, 2002. Private communication.

[Sti88] C. Stirling. A generalisation of Owicki-Gries's Hoare logic for a concurrent while language. *TCS*, 58:347–359, 1988.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambrisge University Press, 2001.

[Wal91] D. Walker. π-calculus semantics for object-oriented programming languages. In T. Ito and A. R. Meyer, editors, *TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.