Failure Mode Assumptions and Assumption Coverage[*]

David Powell
dpowell@laas.fr

LAAS-CNRS
7 avenue du Colonel Roche
31077 Toulouse - France

Research Report 91462

Revized March 30, 1995

Abstract. A method is proposed for the formal analysis of failure mode assumptions and for the evaluation of the dependability of systems whose design correctness is conditioned on the validity of such assumptions. Formal definitions are given for the types of errors that can affect items of service delivered by a system or component. Failure mode assumptions are then formalized as assertions on the types of errors that a component may induce in its enclosing system. The concept of assumption coverage is introduced to relate the notion of partially-ordered assumption assertions to the quantification of system dependability. Assumption coverage is shown to be extremely important in systems requiring very high dependability. It is also shown that the need to increase system redundancy to accommodate more severe modes of component failure can sometimes result in a decrease in dependability.

1 Introduction and Overview

The definition of assumptions about the types of faults, the rate at which components fail and how they may fail is an essential step in the design of a fault-tolerant system. The assumed *type* or nature of faults [6] dictates the type of redundancy that must be implemented within the system (e.g., space or time, replication or diversification,…). Similarly, the assumed *rates* of component failure influence the amount of redundancy needed to attain a given dependability objective. This paper is concerned with assumptions about *how* components fail — i.e., the components' failure modes — *independently of the actual cause or rate of failure*. Assumptions about component failure modes influence both the design and the degree of redundancy of a system's fault-tolerance mechanisms. For example, it is well-known that the tolerance of k faults leading to stopping or "crash" failures [2] requires a redundancy of $k+1$ whereas the tolerance of k faults leading to so-called "Byzantine" failures requires a redundancy of $3k+1$ [5].

[*] This work was partially supported by ESPRIT Basic Research Project nos. 3092/6362: Predictable Dependable Computing Systems

A supposedly fault-tolerant system may fail if any of the assumptions on which its design is based should prove to be false. Indeed, when evaluating the lower bound of the dependability achieved by the system, it must be assumed that the system *will* fail if any assumption made during the system design is violated during system operation. At first sight, this observation favors a design approach based on "worst-case" assumptions since such assumptions are less likely to be violated in the real system. However, as illustrated by the example of "crash" versus "Byzantine" failures given above, when component failure modes become increasingly "arbitrary", the degree of redundancy required to ensure correct error processing must often be increased. Increasing the degree of redundancy not only raises the cost of the system, it also leads to an increase in the possible sources of failure — giving rise to a potential *decrease* in dependability. The designer of a fault-tolerant system is thus faced with the dilemma that although conservative failure mode assumptions are more likely to be true during system operation, the resulting increase in minimum redundancy necessary to enable a proof of correctness of the fault-tolerance mechanisms can lead to an overall decrease in system dependability.

The aim of this paper is to investigate this paradox by establishing a method for formally analyzing the failure mode assumptions on which the designs of fault-tolerant systems are based and for the evaluation of the dependability of systems whose design correctness is conditioned on the validity of such assumptions. The paper is organized as follows.

Section 2 introduces first a model of the service delivered by a system (or a component[1]) to a single user. This model specifies the service as the sequence of value-time tuples of service items that would be perceived by an *omniscient observer* of that service. Formal definitions are then given of various ways in which *individual* service items can be erroneous in the value and time domains. The service model is then augmented to embrace the case of a (single) service delivered to multiple users. Section 3 builds on the definitions of section 2 in order to give formal definitions of the *failure modes* of a system. A failure mode is defined in terms of an assertion on the *sequence* of value-time tuples that a failed or failing system is assumed to deliver. The definitions given in sections 2 and 3 are an extension, and an attempted unification, of concepts set forth by other authors, in particular, those presented in [1-4, 6].

The formalism used for the failure mode assertions of section 3 allows them to be partially ordered by means of an "implication graph". Each path through this graph represents an ordered set of increasingly weaker assertions on system behavior or, equivalently, increasingly severe modes of failure. In terms of fault-tolerant system design, the existence of such orders on the failure modes that can be assumed for the system's components is of very practical significance. First, it enables the design of *families* of fault-tolerance algorithms that can accommodate failure modes of increasing severity. Second, it provides a framework for the analysis of the effect of failure mode assumptions on system dependability. The likelihood of a particular failure mode assertion being true in the real system can be formalized by the *assumption coverage* concept presented in section 4. The coverage of a failure mode assumption is defined as the probability that the assertion that formalizes the assumption is true, conditioned on the fact that the component has failed. In terms of the failure mode implication graph, if assertion X implies assertion Y, then — *for a given component* — the coverage of the failure mode assumption embodied by assertion Y must be greater than that embodied by X since Y is a weaker assertion than X. The directed edges of the implication graph therefore portray increasing assumption coverage.

Section 5 illustrates the effect of failure mode assumptions on system dependability by means of a case study. It is shown that, depending on the coverage of the failure mode assumptions, weaker assertions on component failure modes do not necessarily lead to higher system dependability; in particular, it shows that the weakest design assumption of all — that of

[1] Using the recursive definition of chapter I, a *system* is a set of interacting components and a *component* may be considered as another system. The *service* delivered by a system is its behavior as perceived by the system user - the *user* of a system can be viewed as another system (human or physical) interacting with the former.

arbitrary failures — is not always the best decision for the design of predictably dependable systems.

2 Types of Errors

This section first introduces a model of system service delivered to a *single* user in terms of the sequence of service items that would be perceived by an *omniscient observer* of that service. Formal definitions are then given of various ways in which a *specified* service item can be incorrect in the value and time domains. The set of definitions is then extended to include the case of unspecified or *impromptu* service items that are spontaneously generated by a failed system. An extension of the service model for the case of a service delivered to *multiple* users is then considered.

2.1 Single-user Service

Service Model. The service delivered by a system with a single user can be defined in terms of a sequence of service items, $s_i,\ i=1,2,\ldots$ each characterized by a tuple $\langle vs_i, ts_i \rangle$ where vs_i is the value or content of service item s_i and ts_i is the time[2] or instant of observation of service item s_i.

To define what is a *correct* service item (and from there, what are the different possible sorts of *incorrect* service items), an *omniscient observer* is considered that has complete knowledge of the specified sequence of service items that the system should deliver. For each observed service item, the omniscient observer can decide (a) whether the observed service item was indeed specified and (b) whether its value and time are correct. The omniscient observer can base both decisions (a) and (b) on a value-time domain specification for each service item s_i:

Def. 1. Service item s_i is *correct* iff: $(vs_i \in SV_i) \wedge (ts_i \in ST_i)$ where SV_i and ST_i are respectively the specified sets of values and times for service item $.\, s_i$. ❑

Generally, SV_i and ST_i will be *functions* of the (history of) inputs to the system. However, the definitions given here do not need to refer to the system's inputs.

For many systems, the specified value and time sets are reduced to the special cases $SV_i = \{sv_i\}$ (a single value) and $ST_i = [st_{\min}(i), st_{\max}(i)]$ (a single time interval). Examples of systems where the general case must be considered are: for multiple value sets, one variant of a set of diverse-design software systems and, for multiple time period sets, a system accessing a shared channel by some time-slot mechanism.

A *real* (non-omniscient) observer may not (and usually, does not) have *a priori* knowledge of the specified value-time sets for each service item. Thus, in order for a real observer to detect whether a particular service item is correct in value and time, it must derive the *expected* value-time domains from some other information in its possession. For instance, the "correct" value domain SV_i for a particular service item s_i could be obtained as a function (e.g., a majority vote) of the values of reference service items delivered by a set of similar (redundant) systems. Similarly, the "correct" time domain for a particular service item could be obtained from knowledge of the admissible delay between some reference event and the instant of delivery of the given service item; the reference event could be the delivery of an input to the system, the delivery of a previous service item from the same system or the delivery of one or more service items from other redundant systems. Such practicalities are not of interest when defining — in an absolute manner — what is and what isn't a correct service item. However, they *are* of interest in a practical system when reasoning about the way by which incorrect service items will be perceived (if at all).

[2] "Time" may be measured in seconds, clock ticks, instruction cycles,…; for simplicity, time may be interpreted here as meaning absolute time, although it could equally well be measured with respect to some reference instant.

Errors in a Single-user Service. A system fails whenever any service item is *incorrect*. In the overall service (the *sequence* of service items) delivered by a system, the relationship between various incorrect and correct items of delivered service is captured by the system's *failure mode*. Before defining more formally (in section 3) what is meant by such failure mode assumptions, we shall first consider how *individual* service items can be incorrect. We consider the viewpoint of an enclosing system wherein incorrect service items from a failed component will be perceived as *errors*. According to the definition of a correct service item (cf. definition 1), a distinction can be made between *value errors* and *timing errors*.

Value errors. The definition of a value error is immediately deduced as the corollary of definition 1. The adjective "arbitrary" underlines the fact that the general definition places absolutely no restriction on the erroneous value:

Def. 2. *Arbitrary value error:* $s_i \ : \ vs_i \notin SV_i$ ❑

(The above definition reads: "given service item s_i, the value of s_i (noted vs_i) does not fall within the set of values specified for s_i (noted SV_i)".)

The values of individual service items are often restricted to a sub-set of the universe of values either by some specified syntax for values (e.g., a dictionary of valid symbols) or a range of "reasonable" values. The classic notions of the *theory of error-detecting codes* [19] can thus be taken in a broad sense whereby the set of values that obey a given syntax, or which pass an "acceptance" test, define by extension a set of code values CV, $\left(CV \supseteq SV_i\right)$. Whenever such a "code" exists, it is therefore possible to define a sub-set of value errors called *noncode* value errors:

Def. 3. *Noncode value error:* $s_i \ : \ vs_i \notin CV$ where CV defines a code. ❑

Other authors (e.g., [4]) define the notion of a "null-value" error that is similar in spirit to the above definition of a noncode value error. Whenever the value of a service item is detectably incorrect by simple inspection (i.e., when it does not respect the "code"), an observer can choose to ignore the value or treat it *as if* it was "null". However, in pursuit of precision, the term "noncode value error" is preferred here since there may be several such noncode values and not just one single "null" value.

In [1], a distinction is made between the set of "legal outputs" (legal values) over *all* possible service items delivered by a system and the set of "coarse incorrect outputs" ("unreasonable" values) for a *particular* service item. The latter notion effectively defines a code that is specific to the particular service item based on the knowledge, for example, of previous service items or previous inputs to the system. This distinction is not made here — the important concept behind the notion of noncode values is that they are *detectable* as errors by simple inspection of the value of the service item — irrespectively of how the actual "code" is established.

Timing errors. A timing error is also defined from the corollary of definition 1. An (arbitrary) timing error occurs whenever a service item is delivered outside its specified set of times (this includes the case in which a service item is not delivered at all):

Def. 4. *Arbitrary timing error:* $s_i \ : \ ts_i \notin ST_i$ ❑

Since occurrences in the time domain can, by definition, be ordered, it is possible to distinguish the following classic sub-types for timing errors (e.g., [2]):

Def. 5. *Early timing error:* $s_i \ : \ ts_i < \min\left(ST_i\right)$ ❑

Def. 6. *Late timing error (or performance error):* $s_i \ : \ ts_i > \max\left(ST_i\right)$ ❑

Def. 7. *Infinitely late timing error* or *omission error:* $s_i \ : \ ts_i = \infty$ ❑

An omission error is defined here as a service item that is never delivered, i.e., from the omniscient observer's viewpoint, a service item that is "observed" at time infinity. In [3], an omission is defined as being *either* infinitely late timing *or* a "null-value" (i.e., a noncode value that is "ignored"). This viewpoint is valid when the "path" over which service items are delivered to the user is a dedicated one. However, if this path is shared with other systems and users, the unification of infinitely late timing and ignored noncode values as an "omission" is no longer appropriate. For instance, if the service items under consideration are messages over a bus, there is a fundamental difference between no message being sent and a noncode message being sent. In the former case, the bus remains "free" for another transmission whereas in the second case, the bus is "occupied" (albeit by an incomprehensible and therefore useless message) and is therefore not available for use by other systems.

Impromptu errors. Value errors and timing errors were defined above in terms of deviations of the values and times of a *specified* service item. Failure also occurs when a system spontaneously delivers a service item that was *not* specified — such unspecified service items can be termed *impromptu* errors. Since the correct value and time domains of any service item s_i are given by the set tuple $\langle SV_i, ST_i \rangle$ then an *unspecified* service item s_i is one for which SV_i and ST_i are undefined (noted "$\perp$"):

Def. 8. *Impromptu error:* $s_i \, : \, (vs_i = \perp) \wedge (ts_i = \perp)$ ❏

If the actual value and time of an impromptu service item are considered, then, since the admissible value and time sets are undefined, then *a fortiori*:

$$s_i \, : \, (vs_i \notin SV_i) \wedge (ts_i \notin ST_i)$$

i.e., an impromptu service item is arbitrarily erroneous in both value and time.

From a practical viewpoint, a *real* observer could detect an impromptu error as *either* a value error *or* a timing error, depending on the particular error detection scheme that is used. A real observer (error detection mechanism) would declare an impromptu error to be a value error if it had deduced a value domain SV_j and a time domain ST_j for an expected service item s_j and the impromptu service item s_i falls within ST_j but with $vs_i \notin SV_j$. Similarly, an impromptu error would be declared as a timing error if the impromptu service item s_i was delivered outside any time window in which a service item was expected.

2.2 Multiple-user Service

The service model is now extended to the case of a (single) service delivered to multiple users. In this case, the service is defined as a set of sequences of *replicated* service items. This extension allows the formal definition of the uniformity — or *consistency* — of value-time tuples across the replicated service items.

Service Model. The service delivered by a system to *n* users is a sequence of *n* "replicated" service items, $s_i = \{s_i(1), \ldots, s_i(n)\}, \; i = 1, 2, \ldots$ Each service item "replica" is characterized by a tuple $\langle vs_i(u), ts_i(u) \rangle$, where $vs_i(u)$ is the *value* of replica $s_i(u)$, and $ts_i(u)$ is the time or instant of observation of replica $s_i(u)$. A processor with a private unidirectional multidrop bus is a typical example of such a multiple-user system in which the "users" are the receiving entities on each drop.

In the case of a non-failed system, each user should perceive consistent sequences of replicated service items. In the value domain, this is captured by the constraint: $\forall u \in [1, n], \; vs_i(u) = vs_i$, i.e., all perceived values are equal to some value vs_i. Furthermore, if the set of replicated service item values is to be correct then this value must be within the specified set of values for service item s_i, i.e., $vs_i \in SV_i$.

Since the different users of a multiple-user service are not generally co-located, the instants of delivery of the service items cannot be exactly the same; some other definition of consistency is necessary for the time domain. A definition of consistent instants of occurrence that would appear to be meaningful is that the difference between occurrences should be bounded (on a pair-by-pair basis), i.e., $\forall u,v \in [1,n],\ |ts_i(u) - ts_i(v)| \leq \theta_{uv}$. Furthermore, if the set of service item times is to be correct, then each observed occurrence must lie within a set of times for service item s_i specified for that user, i.e., $\forall u \in [1,n],\ ts_i(u) \in ST_i(u)$.

The considerations discussed above lead to the following definition of correctness for a service item delivered to multiple users:

Def. 9. A replicated service item $s_i = \{s_i(1),\ldots,s_i(n)\}$ is defined to be *correct* iff:

$$\forall u,v \in [1,n],\ \big((vs_i(u) = vs_i) \wedge (vs_i \in SV_i)\big) \wedge \big((|ts_i(u) - ts_i(v)| \leq \theta_{uv}) \wedge (ts_i(u) \in ST_i(u))\big) \quad ❑$$

Errors in a Multiple-user Service. An error occurs in a multiple-user service whenever the assertion in definition 9 is negated. All the definitions of the previous section can be extended to the multiple-user situation by considering errors in individual replicas of a given service item.

However, it is the notion of *consistency* that introduces a new and important dimension to the definition of error types since it allows assertions regarding the *uniformity* of value-time tuples across replicated service items — independently of whether or not the values and times are correct. It is the presence or absence of this uniformity that determines whether or not fault-free multiple users of the service can themselves continue to provide consistent service.

A consistent error occurs in the *value domain* when the value of a replica of a service item is incorrect yet the value consistency condition is respected. If this should occur, then all the replicas of a given service item are identically incorrect:

Def. 10. *Consistent value error:*

$$s_i = \{s_i(1),\ldots,s_i(n)\} \ :\ \forall u \in [1,n],\ (vs_i(u) = vs_i) \wedge (vs_i \notin SV_i) \quad ❑$$

Similarly, a consistent error in the *time domain* occurs when a replica of a given service item is delivered outside its specified set of times yet the overall time consistency condition is respected:

Def. 11. *Consistent timing error:*

$$s_i = \{s_i(1),\ldots,s_i(n)\} \ :\ (\exists u \in [1,n],\ ts_i(u) \notin ST_i(u)) \wedge (\forall u,v \in [1,n],\ |ts_i(u) - ts_i(v)| \leq \theta_{uv}) \quad ❑$$

In the value domain, the notion of coded values allows the definition an interesting "semi-consistent" error scenario in which the values of *some* replicas of a service item are identically incorrect while the other replicas all have *noncode* values:

Def. 12. *Semi-consistent value error:*

$$s_i = \{s_i(1),\ldots,s_i(n)\} \ :\ \forall u \in [1,n],\ \big((vs_i(u) = vs_i) \wedge (vs_i \notin SV_i)\big) \vee (vs_i \notin CV) \quad ❑$$

Semi-consistent value errors can be observed in the practical case of a processor sending coded messages (e.g., with a CRC) over a multidrop bus. The receivers (the users of the service) may receive either incode *or* noncode messages; however, those receivers that receive *incode* messages, receive *identical* value messages. This is a lesser assumption than the atomicity implied by *consistent* value errors.

3 Failure Mode Assumptions

The previous section gave definitions of the ways by which *individual* service items can be correct or incorrect. In this section, we attempt to formalize the concept of an assumed *failure mode* by assertions on the (possibly infinite) sequence of service items delivered by a component. For simplicity, we restrict ourselves to the case of a single-user service and define first assertions that apply independently to the value and time domains. It should be noted, however, that the proposed set of assertions is given as an example; *many* meaningful sets of assertions could be defined.

3.1 Value Error Assertions

Assertions concerning the value of service items are noted V_X where X denotes a particular assertion. Consider the following set of assertions regarding possible value errors produced by a component:

$V_{none} := \forall i,\ vs_i \in SV_i$ ➠ *No value errors occur (every service item is of correct value).*

$V_N := \forall i,\ (vs_i \in SV_i) \vee (vs_i \notin CV)$ ➠ *The only value errors that occur are noncode value errors (every service item value is either correct or noncode).*

$V_{arb} := \forall i,\ (vs_i \in SV_i) \vee (vs_i \notin SV_i) \equiv true$ ➠ *Arbitrary value errors can occur.*

By inspection of the formal assertions, it can easily be seen that they may be represented as nodes of a simple oriented graph (fig. 1) in which an edge directed from node X to node Y means: assertion $X \Rightarrow Y$.

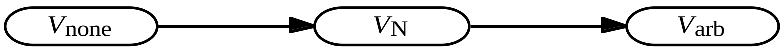

Fig. 1. Value error implication graph

3.2 Timing Error Assertions

Assertions concerning the timing of service items are noted T_X where X denotes a particular assertion. Consider the following set of assertions regarding possible timing errors produced by a component:

$T_{none} := \forall i,\ ts_i \in ST_i$ ➠ *No timing errors occur (every service item is delivered on time).*

$T_O := \forall i,\ (ts_i \in ST_i) \vee (ts_i = \infty)$ ➠ *The only timing errors that occur are omission errors (every service item is delivered on time or not at all).*

$T_L := \forall i,\ (ts_i \in ST_i) \vee (ts_i > \max(ST_i))$ ➠ *The only timing errors that occur are late timing errors (every service item is delivered on time or too late)*

$T_E := \forall i,\ (ts_i \in ST_i) \vee (ts_i < \min(ST_i))$ ➠ *The only timing errors that occur are early timing errors (every service item is delivered on time or too early)*

$T_{arb} := \forall i,\ (ts_i \in ST_i) \vee (ts_i \notin ST_i) \equiv true$ ➠ *Arbitrary timing errors can occur.*

All the assertions defined above effectively define component failure modes that, at the system level, denote *temporary* faults (either intermittent or transient faults). A further assertion of practical interest describes the situation in which a component delivers correctly timed service items up to a particular item and then ceases (omits) to deliver service items. This represents a failure mode that can be viewed at the system level as a *permanent* fault. We shall denote this as *permanent omission* or *crash* and the corresponding formalized assertion is:

$T_P := \forall i,\ (ts_i \in ST_i) \vee (\forall j \geq i, ts_j = \infty)$

A weaker assertion of practical significance corresponds to the situation in which a component may omit to deliver some service items but, if more than k contiguous items are omitted then all further items are omitted. In [18], the term *"bounded omission degree"* is used to characterize such an assumed behavior for a transmission channel. This assumption is formalized by the assertion:

$$T_{B_k} := \forall i,\ \left(ts_i \in ST_i\right) \vee \left(\forall j \geq i, ts_j = \infty\right) \vee \left(\left(ts_i = \infty\right) \wedge \left(\exists j \in \left[i+1, i+k\right]\right),\ ts_j \in ST_j\right)$$

The implication relationships between the various time domain error assertions defined above are represented by the graph of fig. 2.

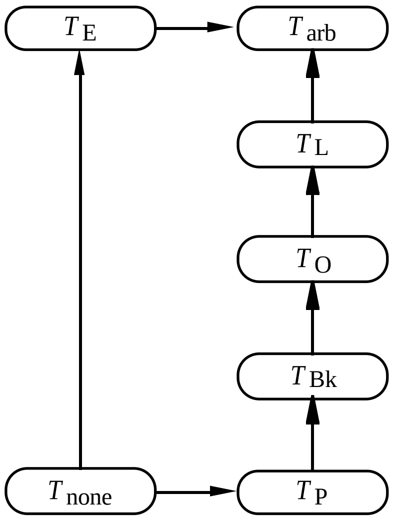

Fig. 2. Timing error implication graph

3.3 Failure Mode Assertions

The complete definition of a component failure mode entails an assertion on errors occurring in both the value and time domains. By taking the Cartesian product of the various definitions given in the two previous sections, 21 conjunctive assertions may be defined; the resulting implication graph is shown in fig. 3.

The implication graph defines a partially-ordered set of assertions regarding the behavior of a component (the implications can be deduced from the formal definitions). The existence of such a partial order means that, if a system's fault-tolerance mechanisms will correctly process errors according to assertion Y, the same mechanisms will be able to process errors according to assertion X if X precedes Y in the partial order. It is thus feasible to design *families* of fault-tolerance algorithms that can process errors of increasing severity [2, 3].

The source node of the implication graph $V_{none} \wedge T_{none}$ designates the strongest assumption that can be made about the behavior of a component, i.e., that the component never induces any errors in the enclosing system, neither in the value domain nor in the time domain.

At the opposite extreme, the sink node of the implication graph $V_{arb} \wedge T_{arb}$ defines the weakest "assumption" that can be made regarding component behavior, i.e., no assumption at all (the assertion $V_{arb} \wedge T_{arb}$ is always true). This means that the component may produce arbitrary errors in both the time and value domains (note that this also includes impromptu errors, cf. §2.1) — such a component may therefore be called a fail-uncontrolled component. It may even be imagined that the behavior of a fail-uncontrolled component is "malicious" in that the timing and the values of service items are malevolently calculated to cause havoc in the system; this viewpoint lies behind the original definition of "Byzantine behavior" [5].

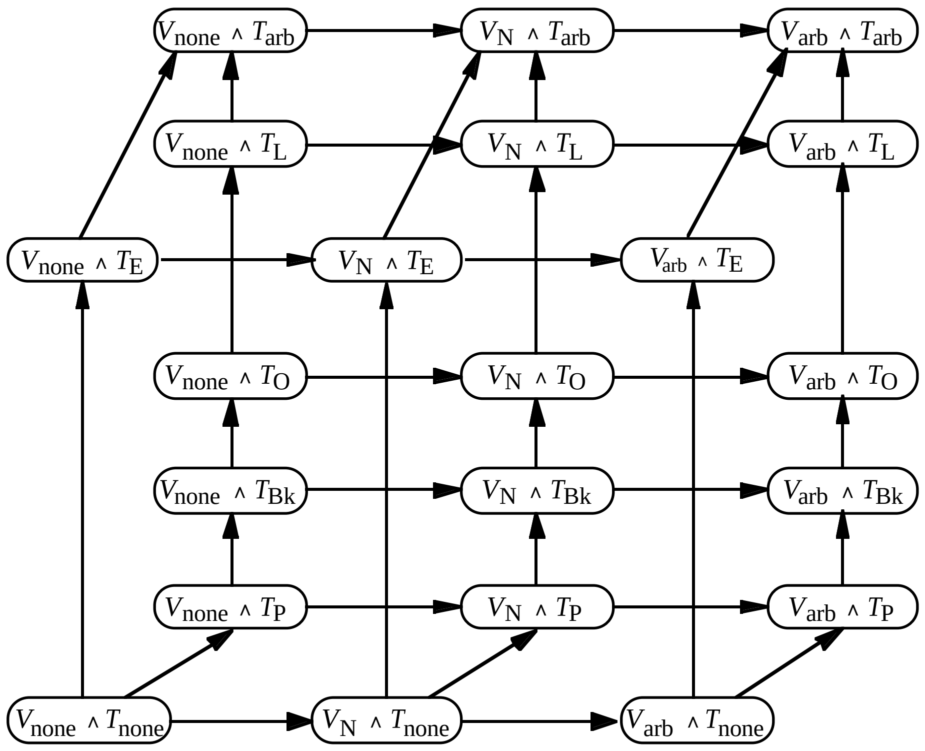

Fig. 3. Failure mode implication graph

Several other nodes in the implication graph are of very real practical significance. The assertion $V_{none} \wedge T_P$ defines the behavior of a fail-silent component [12], i.e., a perfectly self-checking component that, as soon as any fault within the component is activated, the component inhibits all outputs and ceases to provide service items (e.g., messages). From the viewpoint of the other components in the system, a fail-silent component either behaves correctly or remains permanently silent. Such a behavior is sometimes called “crash failure semantics” [2] and represents a sub-set of the properties of the various definitions of “fail-stop processors” [14, 15]. A weaker assertion (called weak fail-silence in [10] is represented by $V_{none} \wedge T_{B_k}$ in which a component may omit to deliver some service items but, if more than k items are omitted, the component will remain forever silent. In both cases, the V_{none} part of the assertion implies that any service items that are delivered are service items with correct content. This has very important implications in the simplification of the design of a system's fault-tolerance mechanisms.

4 Assumption Coverage

The previous section established a set of failure mode assumptions that is partially-ordered due to the implications between the corresponding assertions — leading to failure modes producing errors of increasing “severity”. This section introduces a new concept, called *assumption coverage* [9], that enables another interpretation of such partially-ordered assumptions. It establishes a link between an assumed component failure mode and the dependability of a fault-tolerant system whose design relies on that assumption. The importance of the concept is then illustrated in the next section by a case study.

If the fault-tolerance mechanisms of a system are designed with the assumption that any components that fail will do so according to a given failure mode assertion, then, if any component should fail in such a way that the assertion is violated, the system can fail. In fact, when assessing the dependability of the system, it is preferable to take the pessimistic view that the system *will* fail if a behavior assertion assumed during the system design is violated. In terms of quantitative assessment of dependability, the degree to which a component failure mode assumption proves to be true in the real system can be represented by the notion of the *coverage* of the assumption.

Def. 13. The failure mode *assumption coverage* (p_X) is defined as the probability that the assertion X defining the assumed behavior of a component proves to be true in practice conditioned on the fact that the component has failed:

$$p_X = \Pr\{X = \mathit{true} \mid \text{component failed}\}$$ ❑

The partially-ordered set of assertions defined by an implication graph such as that on fig. 3 defines a set of possible modes of component failure that are of increasing severity. In terms of assumption coverage, the directed edges on the graph can be interpreted as indicating an increase in assumption coverage. Since the assumption of arbitrary timing/value errors $(V_{arb} \wedge T_{arb})$ is — by definition — true, its coverage is equal to 1. Any assumption of a more well-behaved failure mode can only have a coverage less than 1. The strongest assumption of all, that of a *failed* component causing no errors at all $(V_{none} \wedge T_{none})$, cannot possibly be true; its coverage is therefore 0. All intermediate assumptions thus have a coverage $p \in]0,1[$.

Note that the coverage of a given failure mode assumption is different from the coverage of the mechanisms of a fault-tolerant system designed to process errors that occur according to that assumption. The *overall coverage* of a given component failure is given by the product of the coverage of the error-processing mechanisms when errors occur according to a given component failure mode assumption and the coverage of that failure mode assumption, i.e.:

$$\Pr\{\text{correct error processing} \mid X = \mathit{true}\} \times \Pr\{X = \mathit{true} \mid \text{component failed}\}$$

In particular, although the coverage of an assumption of *arbitrary* timing/value errors is equal to 1, this does not mean that the coverage of the *mechanisms* of a fault-tolerant system designed to process such arbitrary errors is necessarily 1. However, it is presumed in the remainder of this paper that the mechanisms of a fault-tolerant system designed to process errors according to a particular failure mode assumption have been “proven” correct, such that the overall coverage is equal to the coverage of the corresponding assumption.

5 Influence of Assumption Coverage on System Dependability: a Case Study

The designer of a fault-tolerant system is faced with the dilemma that although conservative failure mode assumptions (i.e., more severe faulty behavior) will have a higher coverage, there is often an attendant increase in the redundancy that is necessary in order to enable a formal proof of correctness of the system’s fault-tolerance mechanisms. This increase in redundancy can lead to an overall decrease in system dependability. This section is devoted to a simple case study that demonstrates that the designer’s failure mode assumptions should be no weaker than those justifiable by the way that the components are implemented. If this is not the case, the dependability of the system may be less than what might have been achieved with stronger assumptions of less severe faulty component behavior.

5.1 System Definition

Consider a fault-tolerant architecture consisting of n processors each connected to all other processors by a unidirectional multidrop message-passing bus. Each processor carries out the same computation steps and communicates the results of each processing step to all other processors. Each processor applies a decision function to the replicated results received from the other processors to mask errors and triggers other processing steps using the result of the decision function. Each processor must also communicate private information (e.g., local clock values, local sensor information, diagnosis views, etc.) to the other processors; the classical interactive consistency constraints must therefore be respected to ensure that non-faulty processors use such single-source information in a consistent way [5].

Each processor and its associated bus can be viewed as a single component or “fault containment domain” (for the purposes of this case study, we do not consider architectures with multiple types of fault containment domains, e.g., the “inter-stage” technique for interactive consistency [16]). Three possible failure mode assumptions for this composite “processor-bus” component are considered, each leading to a different assumption coverage and requiring a different degree of redundancy to ensure correct error processing in the presence of a given number of faulty components.

Fail-silent Processor-bus. If each message multicast over a processor's bus is accompanied by an error-detection checksum then it is possible to assume that every processor that receives a message with a correct checksum receives the same message, i.e., transmission errors can only lead to semi-consistent value errors (cf. definition 12). If we further assume that any message sent by a processor is a correct message (an important property of the fail-silent processor assumption), then the corresponding assertion on the values of messages delivered by the composite processor-bus component is:

$$V_{FS} = \forall i,\ \left(\forall u \in [1,n],\ \left(vs_i \in SV_i\right) \wedge \left(\left(vs_i(u) = vs_i\right) \vee \left(vs_i(u) \notin CV\right)\right)\right) \tag{1}$$

A fail-silent processor always produces messages on time or ceases to produce messages (forever). Furthermore we assume that, as long as the multidrop bus is intact, it delivers messages to all processors with consistent fixed propagation delays and, if the bus fails, then messages are not delivered to any processor. The corresponding assertion on the timing of messages delivered by the composite processor-bus component is thus that of consistent, permanent omission:

$$\begin{aligned} T_{FS} = \forall i,\ &\left(\left(\forall j \geq i,\ \forall u \in [1,n],\ ts_j(u) = \infty\right) \vee \right. \\ &\left.\left(\forall u,v \in [1,n],\ \left(ts_i(u) \in ST_i(u)\right)\right) \wedge \left(\left|ts_i(u) - ts_i(v)\right| \leq \theta_{uv}\right)\right) \end{aligned} \tag{2}$$

Since any messages delivered with correct checksums are identical, and since any message sent by a fail-silent processor is a correct message, the decision function (in the presence of at most k faults) consists of selecting any message sent from a group of $k+1$ actively replicated processing steps. Furthermore, with semi-consistent value errors, the necessary condition for interactive consistency with k faulty components is $n \geq k+1$ [7]. We postulate that the (strong) assertion T_{FS} allows synchronization of replica execution with $n \geq k+1$ With this pair of value-time failure mode assertions, the system can be configured to be k-fault-tolerant by letting $n = k+1$. Such a k-fault-tolerant configuration will be noted $FS(k)$.

Fail-consistent Processor-bus. The failure mode assertions are now relaxed to allow faulty processors to send erroneous values. However, all messages transmitted are error-checked so that it is still possible to assert that only semi-consistent value errors may occur:

$$V_{FC} = \forall i,\ \left(\forall u \in [1,n],\ \left(vs_i(u) = vs_i\right) \vee \left(vs_i(u) \notin CV\right)\right) \tag{3}$$

Similarly, the time-domain error assertion is weakened to allow consistent timing errors other than consistent omission:

$$T_{FC} = \forall i,\ \left(\forall u,v \in [1,n],\ \left(\left|ts_i(u) - ts_i(v)\right| \leq \theta_{uv}\right)\right) \tag{4}$$

Since any messages delivered with correct checksums are identical, then the necessary condition for interactive consistency with k faulty components is again $n \geq k+1$. However, it is no longer possible to assume that messages with correct checksums are indeed correctly-valued messages (a faulty processor can send an incorrect value but with a correct checksum). The decision function in this case must therefore rely on some sort of majority vote function requiring $n \geq 2k+1$ to be able to mask k faulty values. We postulate that the assertion T_{FC}

allows synchronization of replica execution with $n \geq 2k+1$ With this pair of value-time failure mode assertions, the system can be configured to be *k*-fault-tolerant by letting $n = 2k+1$. Such a *k*-fault-tolerant configuration will be noted $FC(k)$.

Fail-uncontrolled Processor-bus. The value and time-domain assertions are now relaxed to include both arbitrary value errors and arbitrary timing errors, i.e.

$$V_{FU} = \forall i,\ \left(\forall u \in [1,n],\ \left(vs_i(u) \in SV_i\right) \vee \left(vs_i(u) \notin SV_i\right)\right) \tag{5}$$

$$T_{FU} = \forall i,\ \left(\forall u \in [1,n],\ \left(ts_i(u) \in ST_i\right) \vee \left(ts_i(u) \notin ST_i\right)\right) \tag{6}$$

which are both — by essence — identically true.

Since arbitrary value and timing errors can now occur, the interactive consistency constraint is now $n \geq 3k+1$ [5]. The condition for being able to vote on replicated computation ($n = 2k+1$) is therefore also fulfilled. With this pair of value-time failure mode assertions, the system can be configured to be *k*-fault-tolerant by letting $n = 3k+1$. Such a *k*-fault-tolerant configuration will be noted $FU(k)$.

Implication Graph and Assumption Coverages. By inspection of the expressions 1 through 6, the implications between the combined value/time assertions are:

$$V_{FS} \wedge T_{FS} \Rightarrow V_{FC} \wedge T_{FC} \Rightarrow V_{FU} \wedge T_{FU}$$

The coverages $p_{fs},\ p_{fc}, p_{fu}$ of the corresponding assumptions are thus related by:

$$0 \leq p_{fs} \leq p_{fc} \leq p_{fu} = 1$$

In practice, the coverage of a *system-level* assumption about a component's mode of failure can be determined from knowledge about the way by which the *component* has been implemented. For example, if only hardware faults are considered, the claim embodied in assumption V_{FS} can be supported by the use of self-checking hardware and the coverage of assumption V_{FS} can be taken to be bounded by the error-detection coverage that is achievable by such techniques. We will assume that our knowledge about the processor's self-checking mechanisms allows us to place an upper bound on assumption coverage p_{fs} of say 0.99 or 0.999. Assumption V_{FC}, however, is based on the use of message checksums (to ensure that value errors are semi-consistent) and such an error-detection technique is amenable to mathematical demonstration of quite high coverage levels (depending on the ratio of the number of bits in the checksum to the maximum number of bits per message). It would therefore seem reasonable to expect an assumption coverage p_{fc} greater than about 0.9999.

5.2 Dependability Expressions

Markov models of the three architectures $FS(k)$, $FC(k)$ and $FU(k)$ have been established for $k = 1$ and $k = 2$ [11]. Due to space limitations, only the results obtained from the models are given here. The same failure rate λ is used in all three architectures since identical processors are considered — the *only differences* among the architectures result from the different assumptions that are made as to *how* failures manifest themselves.

For the case of systems without maintenance, table 1 gives the expressions for the system reliability *R* in function of the unit reliability $r = e^{-\lambda t}$.

Table 1. Reliability expressions (without maintenance)

***FS*(1)**	$r^2 + 2r\,p_{fs}(1-r)$
***FS*(2)**	$r^3 + 3r^2 p_{fs}(1-r) + 3r\,p_{fs}{}^2(1-r)^2$
***FC*(1)**	$r^3 + 3r^2 p_{fc}(1-r)$
***FC*(2)**	$r^5 + 5r^4 p_{fc}(1-r) + 10r^3 p_{fc}{}^2(1-r)^2$
***FU*(1)**	$r^4 + 4r^3(1-r)$
***FU*(2)**	$r^7 + 7r^6(1-r) + 21r^5(1-r)^2$

When systems with maintenance are considered, it is assumed that the repair time is exponentially distributed with rate μ. Table 2 gives the expressions of the “equivalent failure rate” λ_{eq} [8] for the single repairman case when $\lambda/\mu << 1$ (i.e., the mean time to repair is much smaller than the mean time to failure). The equivalent failure rate can be used to express the system reliability (with maintenance), $R' = e^{-\lambda_{eq}t}$ and the asymptotic system availability, $A = 1 - \lambda_{eq}/\mu$.

Table 2. Equivalent failure rate expressions (with maintenance such that $\lambda/\mu << 1$)

***FS*(1)**	$2\lambda\left[\left(1-p_{fs}\right) + p_{fs}\lambda/\mu\right]$
***FS*(2)**	$3\lambda\left[\left(1-p_{fs}\right)\left\{1+2\left(p_{fs}\lambda/\mu\right)\right\} + 2\left(p_{fs}\lambda/\mu\right)^2\right]$
***FC*(1)**	$3\lambda\left[\left(1-p_{fc}\right) + 2\left(p_{fc}\lambda/\mu\right)\right]$
***FC*(2)**	$5\lambda\left[\left(1-p_{fc}\right)\left\{1+4\left(p_{fc}\lambda/\mu\right)\right\} + 12\left(p_{fc}\lambda/\mu\right)^2\right]$
***FU*(1)**	$12\lambda(\lambda/\mu)$
***FU*(2)**	$210\lambda(\lambda/\mu)^2$

We shall now consider the various architectures in two different applications: a) a life-critical application without maintenance, and b) a money-critical application with maintenance.

5.3 A Life-critical Application

Consider a system reliability objective of $R \geq 1 - 10^{-9}$ over 10 hours (corresponding to a typical civil aviation requirement for a fly-by-wire flight control system) and consider that the reliability of a single processor is given by $r = e^{-\lambda t}$ with $1/\lambda = 5$ years. Fig. 4 gives the unreliability $UR = 1 - R$ at 10 hours plotted as a function of assumption coverage p for a non-redundant system (noted NR) and the 1- and 2-fault-tolerant configurations.

It can be seen that configuration $FU(2)$ allows the 10^{-9} objective to be attained. However, configurations $FC(2)$ and $FS(2)$ also allow the objective to be attained *if* the coverage of the assumptions on which they are based is greater than 0.999999. Now, although this is not a reasonable value for p_{fs}, it does seem possible for p_{fc} (cf. §5.1). In this case, the $FC(2)$ configuration with 5 processors can be considered a viable alternative to the $FU(2)$ configuration with 7 processors. If the coverage p_{fc} could be verified to be even higher, then fig. 4 shows that $FC(2)$ is better than $FU(2)$ even though worst-case (Byzantine) failures were not assumed in its design.

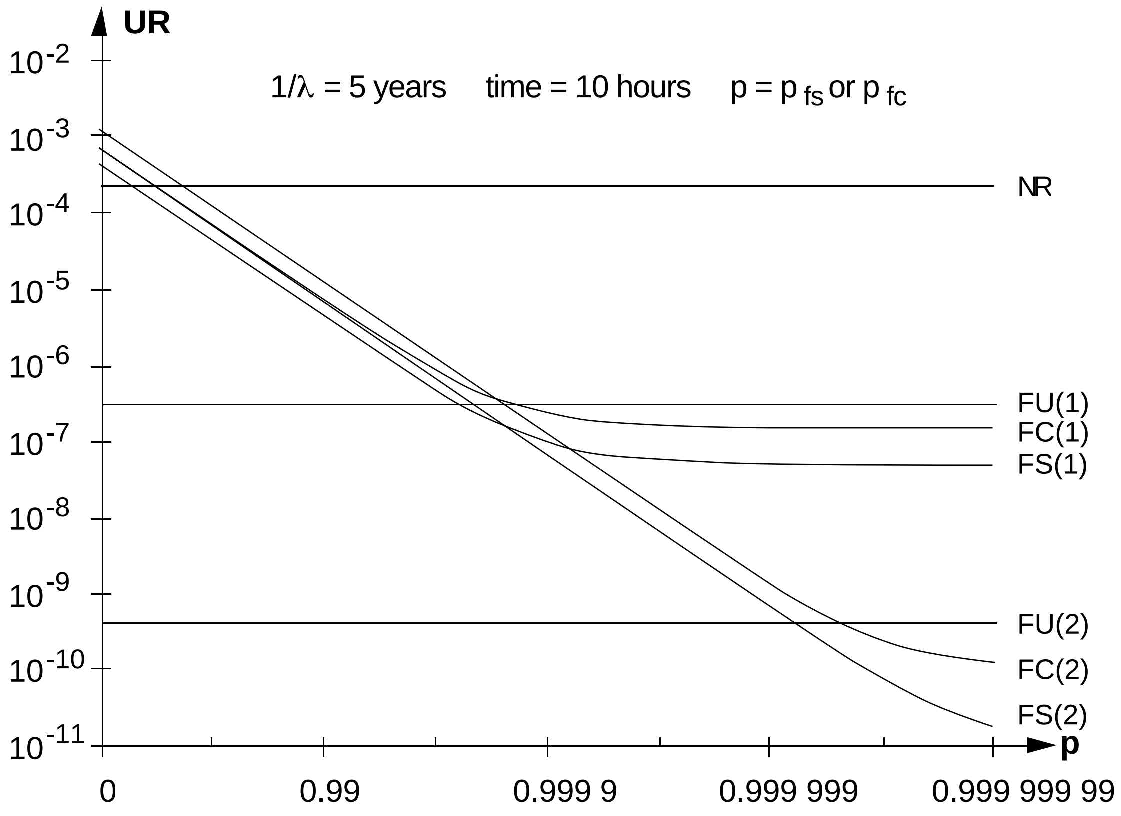

Fig. 4. Unreliability versus coverage (without maintenance)

Careful inspection of fig. 4 also reveals that, for assumption coverages less than about 0.9995, configuration $FS(2)$ is less reliable than configuration $FS(1)$ although it is capable of tolerating two simultaneous faults instead of just one. The same is true for configuration $FC(2)$ with respect to $FC(1)$. This is an illustration of the well-known "point of diminishing returns" [17] that occurs with less-than-unity coverage — a degree of redundancy exists beyond which any further increase is detrimental to dependability.

Fig. 4 illustrates that, depending on the assumption coverage, the reliability achieved by solutions with restrictive failure mode assumptions can be higher than that in which less restrictive assumptions are made — including the 100% coverage case of assuming worst-case failure modes. Like the "diminishing returns" phenomenon mentioned above, this is due to the higher redundancy needed to process more complicated errors (leading to an increase in the possible sources of system failure).

5.4 A Money-critical Application

Consider now an application in which maintenance can be carried out and for which the dependability criterion is availability rather than reliability (for example, an electronic switching system). Fig. 5 gives the unavailability $UA = 1 - A$ plotted as a function of assumption coverage p for the non-redundant system NR and the 1- and 2-fault-tolerant configurations with $1/\lambda = 5$ years and $1/\mu = 1$ day.

The curves on fig. 5 show that unavailability can be less than 6 minutes per year with the dual-redundant configuration $FS(1)$ (the cheapest fault-tolerant configuration) with an assumption coverage p_{fs} of only 99% — which seems reasonable (cf. §5.1). The QMR configuration

$FU(1)$, tolerating the worst-case failure mode, achieves an unavailability of about 1 minute per year whereas the TMR configuration $FC(1)$, with an assumption coverage p_{fc} of 0.9999, allows the yearly unavailability to be decreased to less than about 45 seconds per year. If even lower unavailability is required then a configuration tolerating two faults must be used. Note, however, that configuration $FC(2)$ enables a lower unavailability (less than 1/3 second per year) than configuration $FU(2)$ if the assumption coverage p_{fc} is greater than about 0.99999 (such high figures may not be reasonable however for p_{fs}, cf. §5.1, so the theoretical best, $FS(2)$, may not a practical option).

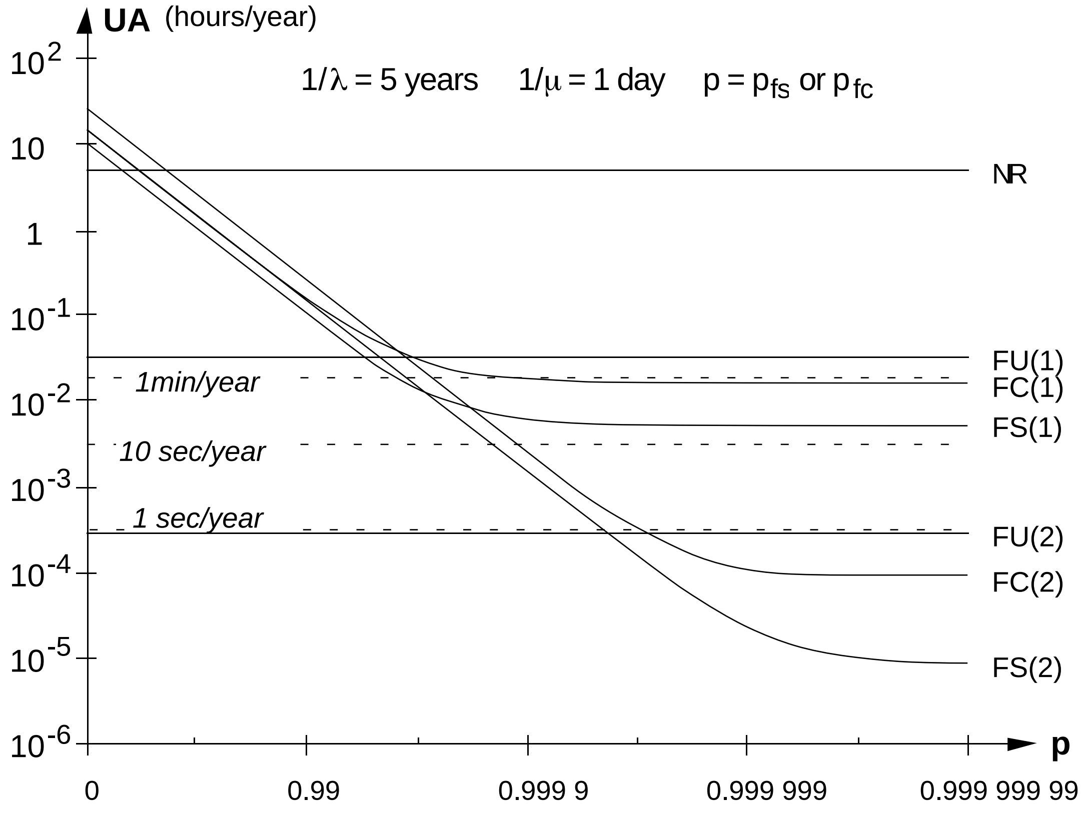

Fig. 5. Unavailability versus coverage

Fig. 6 gives the unavailability of the various configurations as a function of mean repair time ($1/\mu$) when a particular set of (reasonable) values for p_{fs} and p_{fc} are taken, viz. $p_{fs} = 0.99$ and p_{fc} = 0.9999. The mean time between processor failures ($1/\lambda$) is again taken as equal to 5 years.

Let us suppose now that the objective is to obtain an unavailability of less than 1 minute per year and consider three repair scenarios:

1) repair can be carried out, on average, within the *hour* (e.g., by personnel located on site); the most cost-effective configuration would be the dual-redundant configuration $FS(1)$ tolerating a single fault conditioned on it leading to a "silencing" failure mode (note again, that due to the diminishing returns phenomenon, the addition of an extra active spare, $FS(2)$, is in fact detrimental to availability),
2) repair can be carried out, on average, within a *day* (e.g., by personnel urgently dispatched to the site); the most cost effective solution would be a TMR system $FC(1)$ tolerating a single fault conditioned on it leading to a consistent failure mode,

3) repair can only be carried out, on average, within a *week*; there is no 1- or 2-fault-tolerant solution — however, the best possible solution (leading to an unavailability of about 3 minutes per year) would be the $FC(2)$ configuration (a 5 processor system tolerating 2 faults conditioned on them leading to a consistent failure mode); note that, even though configuration $FU(2)$ can tolerate two faults that lead to arbitrary failures, the long repair time considered here means that it is no longer the optimum solution.

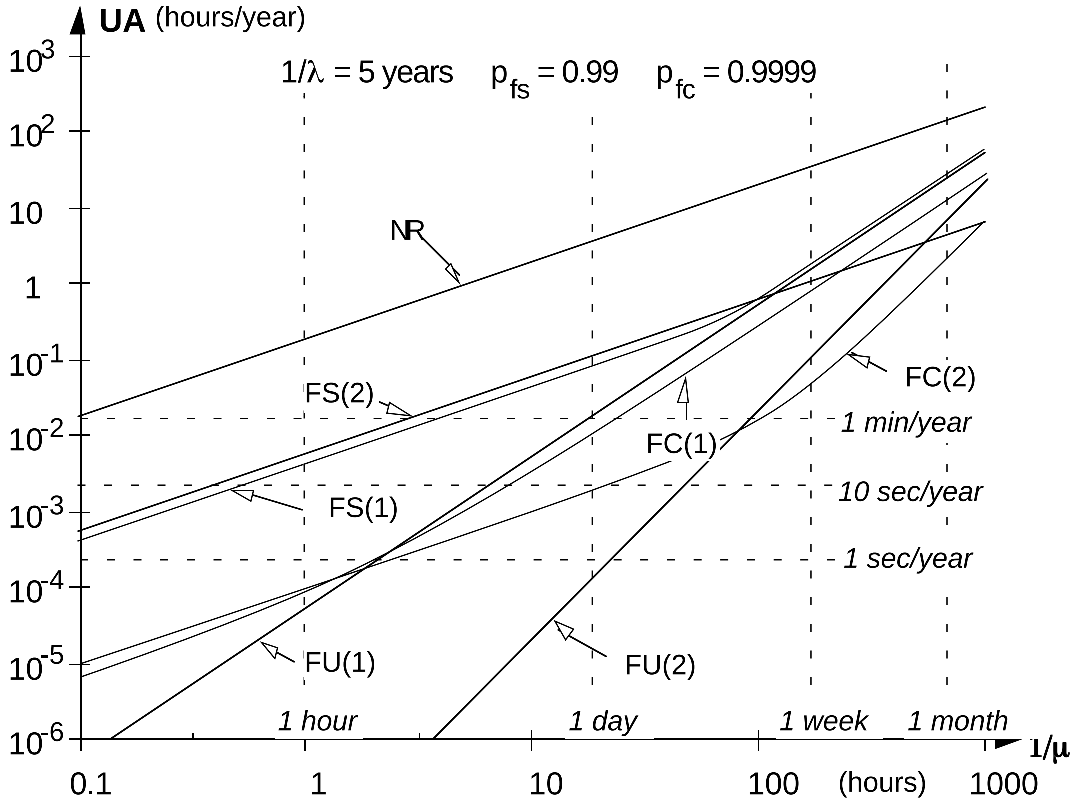

Fig. 6. Yearly unavailability versus repair time

6 Conclusions and Future Directions

This paper has attempted to provide a formalism for describing component failure modes in terms of the sequences and multiplicity of value and timing errors that a failed component may introduce into a system. The link between partially-ordered failure mode assertions and system dependability has been clarified by the notion of "assumption coverage". The importance of the latter has been underlined by means of a simple case study.

As stated in section 3, there are numerous possibilities for defining partially-ordered sets of value/timing error assertions and thus much scope remains for extension. In particular, the multiple-user error types defined in section §2.2, which include the important notion of consistency, introduce a new dimension to the definition of partially-ordered sets of failure modes that has not yet been fully explored.

A further extension of practical interest would be to include the concept of "fairness" that is necessary to prove the termination of protocols employing time-redundancy for ensuring delivery of messages over transmission channels that "lose" messages [13]. The assertion $V_{none} \wedge T_O$ (the "omission failure" semantics of [2]) does not portray a "fair" channel since it states that any *or all* service items may be omitted. On the contrary, a fair channel guarantees (with probability 1) that at least one item of an infinite sequence will be delivered.

Another area where further research would be useful is in the definition of necessary timing error assumptions in the design of fault-tolerant protocols. In section §5.1, we postulated that the timing assertions given in expressions (2) and (4) were sufficient to allow fault-tolerant synchronization with $k+1$ and $2k+1$ processors. Proof of these postulates (or similar) would be of interest.

Finally, it should be noted that the assumption coverage notion can be extended to any sort of assumption made during a system design (i.e., not just assumptions about component failure modes). For example, it could also be used to embrace assumptions about the peak-load submitted to a real-time system or about the independence of faults in redundant units.

Acknowledgment: My particular thanks go to Jean-Claude Laprie with whom I have had many fruitful discussions that have helped no end in the clarification of the concepts set forth in this paper. I would also like to acknowledge the useful comments and ideas from discussion with or private communication from Tom Anderson, Jean Arlat, Paul Ezhilchelvan, Jack Goldberg, Brian Randell, Santosh Shrivastava, Luca Simoncini and Paulo Veríssimo.

References

[1] A. Bondavalli and L. Simoncini, *Failure Classification with respect to Detection*, Specification and Design for Dependability, Esprit Project N°3092 (PDCS: Predictably Dependable Computing Systems), First Year Report, May 1990.

[2] F. Cristian, H. Aghali, R. Strong and D. Dolev, “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement”, in *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15),* (Ann Arbor, MI, USA), pp.200-206, IEEE Computer Society Press, June 1985.

[3] P. D. Ezhilchelvan and S. K. Shrivastava, “A Characterization of Faults in Systems”, in *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems,* (Los Angeles, CA, USA), pp.215-222, IEEE Computer Society Press, January 1986.

[4] P. D. Ezhilchelvan and S. K. Shrivastava, *A Classification of Faults in Systems*, University of Newcastle upon Tyne, UK, Technical Report, 1989.

[5] L. Lamport, R. Shostak and M. Pease, “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems,* 4 (3), pp.382-401, July 1982.

[6] J.-C. Laprie (Ed.), *Dependability: Basic Concepts and Terminology,* Dependable Computing and Fault-Tolerance, 5, 265p., Springer-Verlag, Vienna, Austria, 1992.

[7] W. R. Moore and N. A. Haynes, “A Review of Synchronisation and Matching in Fault-Tolerant Systems”, *Proc. of the IEE,* E-131 (4), pp.119-124, July 1984.

[8] A. Pagès and M. Gondran, *System Reliability: Evaluation and Prediction in Engineering,* Springer-Verlag, New York, USA, 1986.

[9] D. Powell, “Fault-Tolerance in Distributed Systems: Error Assumptions and their Importance”, in *Proc. Franco-Brazilian Seminar on Distributed Computing Systems,* (Florianópolis, SC, Brazil), pp.36-43, Federal University of Santa Catarina, September 1989 (in French).

[10] D. Powell (Ed.), *Delta-4: a Generic Architecture for Dependable Distributed Computing,* Research Reports ESPRIT, 484p., Springer-Verlag, Berlin, Germany, 1991.

[11] D. Powell, *Fault Assumptions and Assumption Coverage,* Esprit Project N°3092 (PDCS: Predictably Dependable Computing Systems), Second Year Report, May 1991.

[12] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck, “The Delta-4 Approach to Dependability in Open Distributed Computing Systems”, in *Proc. 18th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18),* (Tokyo, Japan), pp.246-251, IEEE Computer Society Press, June 1988 (reprinted in Readings in Ultra-Dependable

Distributed Systems, Edited by N. Suri, C.J. Walter and M.M. Hugue, IEEE Computer Society Press, 1994).

[13] J. P. Queille and J. Sifakis, “Fairness and Related Properties in Transition Systems: a Temporal Logic to deal with Fairness”, *Acta Informatica*, 19 (3), pp.195-220, 1983.

[14] R. D. Schlichting and F. B. Schneider, “Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems”, *ACM Transactions on Computer Systems*, 1 (3), pp.222-238, August 1983.

[15] F. B. Schneider, “Byzantine Generals in Action: Implementing Fail-Stop Processors”, *ACM Transactions on Computer Systems*, 2 (2), pp.145-154, May 1984.

[16] T. B. Smith, “High Performance Fault-Tolerant Real-Time Computer Architecture”, in *Proc. 16th Int. Symp. on Fault-Tolerance Computing (FTCS-16)*, (Vienna, Austria), pp.14-19, IEEE Computer Society Press, July 1986.

[17] J. J. Stiffler, “Fault Coverage and the Point of Diminishing Returns”, *Journal of Design Automation and Fault-Tolerant Computing*, 2 (4), pp.289-301, October 1978.

[18] P. Veríssimo, “Redundant Media Mechanisms for Dependable Communication in Token-Bus LANs”, in *Proc. 13th Local Computer Network Conf.*, (Minneapolis, MN, USA), pp.453-462, IEEE Computer Society Press, October 1988.

[19] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North-Holland, New York, 1978.