Component Failure Mitigation According to Failure Type

Fan Ye, Tim Kelly
Department of Computer Science, The University of York, York YO10 5DD, UK
{fan.ye, tim.kelly}@cs.york.ac.uk

Abstract

Off-The-Shelf (OTS) software components are being used within complex safety-critical applications. However, to use these untrustworthy components with confidence, it is necessary to ensure that potential failures of the components cannot contribute to system level hazards. This requires the system level effects of component failures to be understood and mitigated using suitable fault tolerance techniques. However, the black-box nature of an OTS component implies the visibility and modifiability of the component is very limited. This restricts the choice of available fault tolerance techniques in mitigating failures of an OTS component. This paper presents a systematic approach to facilitate the selection of appropriate mitigation strategies according to a classification of failure types of an untrustworthy component. This approach enables an untrustworthy component to be used in a safety-critical context with increased confidence.

1. Introduction

OTS software components are increasingly used within complex safety-critical software systems. This is because of the perceived potential savings on development cost and time. However, in order to achieve its overall safety objective, a safety-critical system often requires its constituent components to be of high dependability. Due to the black-box nature of an OTS component, evaluation techniques that can be applied to the component are often limited. This means adequate assurance about the quality of such a component cannot be gained through evaluation. Therefore, in order to use such untrustworthy components with confidence, it is necessary to ensure that potential failures of the components cannot contribute to system level hazards.

Design faults remaining in an OTS component cannot be dealt with by its acquirer. This is because the process of error identification, fault location and fault removal requests the access to the component's source code that is often unavailable. However, the failures of such a component *are* identifiable and their impact onto a system (integrated with the component) *is* analysable. Those critical component failures can then be dealt with by means of fault tolerance techniques devised at the system context.

Not all fault tolerance techniques are suitable for addressing failures of a black-box component (due to the need for some techniques to have access to internal component details). For instance, exception handling is a well-known technique for fault tolerance, but it cannot be added into a black-box component by the acquirer without access to the source code.

We believe that different types of component failures should be dealt with differently. In this paper, we describe a systematic approach to facilitate the selection of appropriate mitigation strategies according to a taxonomy of component failure types.

2. Software fault tolerance

2.1. General principles

The concepts of fault, error, and failure are directly linked in a causal relationship: faults lead to errors, which ultimately lead to failures. A fault is a physical defect or design flaw within a hardware or software component. An error is the manifestation of a fault. A failure is an externally observable event representing a deviation from the authoritative service specification [8].

There are two ways to tackle faults: fault prevention and fault tolerance. Fault prevention is achieved through avoiding faults introduction during production (fault avoidance) or/and removing them before deployment (fault removal) [4]. Fault tolerance aims to provide correct function despite the presence of faults [9].

Fault prevention has little to do with an OTS component once it is delivered to its acquirer. The black-box nature often means that modification to the

component even for the purpose of correcting an identified fault can only be carried out by its provider. Moreover, despite the efforts of fault avoidance and fault removal, fault free software cannot be achieved [4]. Therefore, for critical systems whose failures could mean loss of human lives, environmental damages, and huge financial losses, it is imperative for them to prevent such failures from occurring in presence of those residual faults. Fault tolerance is left as the only means to protect a system from the failures of an integrated untrustworthy component.

There are four constituent phases of fault tolerance [4]: error detection, damage assessment, error recovery and fault treatment.

The failure of a software component does not necessarily lead to system failure. However, a component failure may introduce errors into the system state, and if this has not been identified and dealt with, the system can fail as a result.

2.2. Component failures identification

The failures (or more specifically, **failure mode** – the effect by which a failure is identified [11]) of an untrustworthy component within a safety-critical application are to be identified before they can be dealt with. The failure modes for an OTS component can be identified from two sources: field experience, and deviation-based analysis over the functionality provided by the component.

Field experience (if exists) of an OTS component builds up a failure profile according to users' feedback. Nevertheless, some failures will not have been identified this way. Deviation-based analysis prompts consideration for all plausible deviations that can occur to the functions provided by a component. This is pessimistic, as some identified failures cannot occur due to the component's internal implementation.

Fault tree analysis, through establishing a causal relationship between high-level system hazards and component failures, can highlight individual failures whose impact on system safety are more significant than those of other failures. Those failures with high system safety impact are to be addressed with higher priority.

2.3. Possible mitigations for failures of untrustworthy components

Over the years, a number of fault tolerant techniques ranging from simple sanity check to complex N-version programming have been established. Major fault tolerance techniques are summarised as follows:

Sanity checks – also known as reasonableness checks [4], this technique uses known semantic properties of data (e.g. range, rate of change, and sequence) to detect errors.

Reversal checks – reversal checks use the output of a module to compute the corresponding inputs based on the function of the module [4]. An error is detected if the computed inputs do not match the actual inputs.

"Control + Monitor" – a simplest form of design diversity, which involves using a control component and a monitor component [2]. The monitor component implements a much simpler logic to provide a degraded service compared to the complex service provided by the control component. If the outputs from the two components are equivalent within some tolerance, the results from the control component are deemed correct and used by the system; otherwise the results from the monitor component will be used.

Recovery blocks – with the recovery blocks technique, multiple versions of a software module or component (a primary and one or more alternate modules) are sequentially executed, until either an acceptable output is obtained or the alternates are exhausted [10]. An acceptance test is used for error detection. It is not necessary for all the modules to produce exactly the same results, as long as the results are acceptable as defined by the acceptance test.

N-version programming (NVP) – this scheme involves using N (N>2) versions of independently implemented software components to provide the same services [1]. A majority voting mechanism is implemented to detect and mask output errors from faulty component(s).

Watchdog timer – watchdog timer is commonly used to deal with failures related to service timing. This requires the process being monitored to reset the timer before it expires as an indication that the process is operating satisfactorily [10]. It can detect whether a task overrun its scheduled processing time, and the absence of outputs from an untrustworthy component. Where the absence of some output is deemed hazardous, a watchdog timer could be used to detect the absence of the component output and to trigger the use of some other logic (e.g. monitor component) to produce an alternative output.

Timestamp – through attaching some timing information to each of a sequence of messages to indicate the correct ordering, this technique can be used to deal with a sequence of messages arriving in wrong order. However, this approach may not work for black-box components. For example, out-of-sequence messages originated from a black-box component cannot be dealt with by means of timestamp technique if this technique has not already been implemented into the component.

Coding checks – sometimes high assurance data needs to be handled by an untrustworthy component without changing its contents. In order to ensure the data integrity, Message Authentication Codes (MACs) (e.g. checksums, cyclic codes) could be applied to the data and checked by the application.

Wrapper – wrapping is not a specific fault tolerance technique. A wrapper is a specialised component inserted between a third-party component and its application environment to deal with the flows of control and data going to and/or from the component [5]. Different mitigation strategies can be implemented into a wrapper to deal with different types of failures that may arise from an untrustworthy component.

Design diversity (e.g. "Control + Monitor", Recovery Blocks, NVP) which employs multiple channels to provide the same or degraded services is a powerful means to protect system from failures of an untrustworthy component. In this scheme, the untrustworthy component is one of the channels used to provide the required service. If two channels are used, the other channel can be an in-house implemented high assurance component to provide some degraded core services should failures of the untrustworthy component occur. Three or more channels (either third-party or in-house developed components) may be used to mask failures of the untrustworthy component (e.g. Diversity of Off-The-Shelf components [6]). However, the design diversity scheme can be very expensive in terms of cost and time of developing or acquiring some other component(s).

We believe that a thorough understanding on the nature of different types of component failures may offer some insights on how those failures can be addressed cost-effectively by individual or combination of appropriate fault tolerance techniques.

3. Component failures classification

Close examination of different types of software component failures may offer some insights on how these failures can be detected and their effects mitigated.

A number of classification schemes for software failure types have been proposed in literature. In particular, the one proposed by Pumfrey [7] was selected as the most appropriate for the approach described in this paper. This is because of its emphasis on the detectability of failures, a critical point when considering strategies for handling failures. This classification of failure type is based on the notion of the system as the provider of a *service*, or a *sequence of services* [7]. Each service consists of a particular value, delivered within a defined interval. Six failure types are defined with respect to service provision, service timing and service value:

- **Commission** – service delivered when not required.
- **Omission** – no service is delivered.
- **Timing** – the service delivered earlier/later than intended. This may be absolute (i.e. early/later compared to a real-time deadline) or relative (i.e. early/later with respect to other events or communications in the system).
- **Value** – a service is delivered within the correct interval, but with an incorrect value.

It is recognised that there exists close relationship among the different failure types [3]. Timing and value failures are two more general types of component failures. One failure may involve both service timing and value (i.e. a service is delivered outside the correct interval and with an incorrect value.). Omission is a special case, or proper subset, of timing or value type failures. For example, omission can be regarded as either a timing failure, where the delivery time is infinitely late, or a value failure where the value delivered is null. Likewise, commission can be regarded as either a timing failure, a value failure, or a time and value failure.

4. Mitigation by component failure types

In this section, different types of component failures are represented using Unified Modelling Language (UML) sequence diagrams, and their interpretations and potential means for detection and mitigation are discussed in turn (Sections 4.1 – 4.4). A brief summary is presented in Section 4.5.

4.1. Timing failures

Representation, interpretation, and examples

A timing failure occurs when the return of a procedure call missed its deadline (T_1 in Figure 1), or a message is sent/read by the component at a time that is outside the expected time interval (T_2 in Figure 1), or a sequence of messages are sent in a wrong order (T_3 in Figure 1).

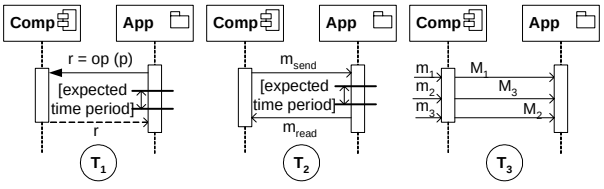

Figure 1. Early/Late type failure scenarios

In time related failure scenario 1 (i.e. the sequence diagram T_1), the return of a requested procedure call[1] missed its deadline. This can happen if the component service has not been designed to meet the time constraints as specified by the application, or the normal operation has been disrupted in some way (e.g. waiting for resources). This failure scenario can be regarded as an omission type failure, as the required service was not delivered by the specified deadline.

In time related failure scenario 2 (i.e. the sequence diagrams T_2), a message passing initiated by the component occurs outside the expected time interval, either earlier or later. A message may be sent into the communication channel earlier or later than expected. This can potentially result in an unread message within the communication channel being overwritten (in the case of early message sending), or an old message being read and dealt with by the application again (in the case of message being sent late). Similarly, a message may also be read from the communication channel by the component earlier or later than expected. Likewise, an old message may be read (early message reading), or unread message be overwritten (late message reading).

In time related failure scenario 3 (T_3 in Figure 1), a sequence of messages sent from the component are in an incorrect order (with respect to the order of the incoming messages – e.g. received from system environment through sensors). For example, in an image processing system where a sequence of transformations are carried out to continuously produce pictures from raw image data, a third-party component is in charge of one step of the image transformation, and the sequence of messages sent out by the component (e.g. packets of image data) may be incorrect due to some internal faults, which could result in the final constructed image being incorrect.

Detection and mitigation

The first two cases of timing failures (T_1 and T_2 in Figure 1) can be detected by some forms of time monitor such as a watchdog timer. If timing constraints are specified for the completion of a component procedure call, and for sending or receiving a message, the application knows when this should happen, therefore a watchdog timer can be implemented to indicate the timing failure if the timing constraints are violated.

As for mitigation, in the first and second (the part denoting late message sending/reading) scenarios, if the component service has not been implemented to meet the required timing constraints, it should not be used; otherwise the watchdog timer should indicate timing failure at some time threshold (prior to the arrival of the deadline) so that the required service can be provided by some other module. In systems where the timing constraints are so tight that consecutive execution of two components providing the same required service is infeasible, parallel execution of the two components can be used. As such failures are often transient (e.g. time or configuration dependent), concurrent execution of two copies of the same version software component may well be able to cope with these timing failures.

The timing failure in the second scenario (the part denoting early message sending/reading) can be mitigated by means of synchronisation (i.e. blocking any component-initiated message passing that occurs earlier than expected) upon failure detection.

The connector may also have a role to play on the mitigation of the second failure scenario. The behaviour of the connector can be constrained such that overwriting existing unread messages and reading old messages are not allowed.

As for the failure depicted in the third scenario, if ordering information (e.g. order id, timestamp) can be attached to the incoming messages and carried through to the outgoing messages, the incorrect messages ordering can be easily detected and corrected. In cases where the relationship between an incoming message and an outgoing message is one to one, a simple comparison of incoming and outgoing messages' content would reveal any incorrect ordering. However, in a more general scenario where the sequence of messages sent from the component do not depend upon the existence of the same number of related incoming messages, the failure may only be detected and corrected by using multiple parallel channels.

4.2. Value failures

Representation, interpretation, and examples

A value failure occurs when the component produces incorrect outputs in response to a procedure call or sends out an asynchronous message with incorrect content. These two component value failure scenarios are illustrated in Figure 2.

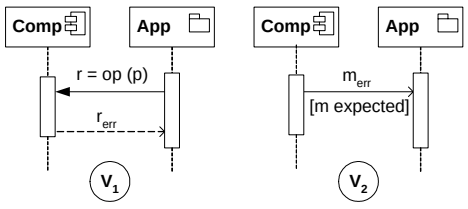

Figure 2. Value type failure scenarios

[1] r = op (p), denotes a method op is called with some parameter p and return value r

In value related failure scenario 1 (V_1 in Figure 2), the component produced incorrect outputs (i.e. r_{err}) for a requested operation. The outputs are not simply the return values of a procedure call, but including all the changes made to the state of the application during the method invocation. This is because the return values of many procedure calls are merely an indication of the completion of the method invocation. For example, the return value (e.g. an integer 0) for the method call removing an employee's record from a database tells us nothing but the invocation completed successfully, and we do not know which record was actually removed from the database.

In value related failure scenario 2 (V_2 in Figure 2), the content of a message sent from the component is incorrect. The message content may be incorrect with respect to the format, parameters' range and so on.

Detection and mitigation

Value failures can only be detected through comparing the actual outputs from an operation with the expected ones (if known) or with the outputs from an alternative module. The simplest form of such comparison would be the reasonableness check, for example, simply check if the message received is within a specified range and of the required format.

As for the case of a procedure call, if it only involves producing some results from available information without modifying the system state, reasonableness check can also be applied to identify some simple value failures, or a reversal check can be carried out to detect value failures.

In a more general form of procedure call that involves producing some results as well as modifying system state, failure detection will have to check if both the return values and the state changes are as specified. For this purpose, system state just prior to the invocation of a procedure call should be the recorded. The pre- and post-conditions of the procedure call can be used to infer the expected system state after the invocation, this expected system state can then be compared with the actual post invocation system state to identify if any value failure has occurred. For example, removing a data record from a database would request the record to be present before method invocation, and after the invocation, the number of data records would be one fewer and the record being removed should not be in the database; comparison would reveal if any value failure has occurred. Upon failure detection, actual mitigation would involve restoring the system state to that of the pre-invocation, and use an alternative module to provide the same service (e.g. recovery blocks).

Using multiple channels to deliver the same service is a more powerful means of value failure detection and mitigation. "Control + Monitor", recovery blocks and NVP can all be employed to efficiently detect value failures on an untrustworthy component and provide corrective actions or mask the failures through majority voting.

In systems where an untrustworthy component involves generating outputs from high critical data (e.g. the rail network mapping data within a railway signalling system) without modifying its contents, protecting the data from being unexpectedly modified by operations of the untrustworthy component may simply mitigate a whole class of value failures. Data protection can be done by means of attaching some MAC such as a checksum to each data item.

4.3. Commission failures

Representation, interpretation, and examples

A commission failure occurs when a service is provided when not required. In a component-based system, such failures of a component would appear in one of the two forms: an unexpected activation of an operation within the component, sending or reading an unexpected message through a communication channel. These two forms of component commission failures are illustrated as three sequence diagrams in Figure 3.

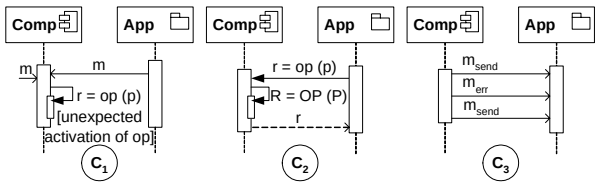

Figure 3. Commission type failure scenarios

There is always some stimulus that triggers the unexpected activation of an operation. The stimulus could be an asynchronous message m sent from the operational environment, from another component within the application package (as in C_1 of Figure 3), or a method call requested by another system component (as in C_2 of Figure 3). The sequence diagram C_3 represents the other form of component commission failure (i.e. sending or reading an unexpected message m_{err}).

Sequence diagram C_1 illustrates the scenario of unexpected activation of an operation while the component is dealing with some legitimate incoming message. This operation is not required for handling the message. It may be one of the functions that is not used by the specific application at all (e.g. an Easter egg). The execution of such an operation may or may not cause changes to the system state. For example, a function that simply calculates the sum of all

employees' salary will not change the internal state of a system; on the other hand, a function that removes an employee's record from the database, if activated accidentally, will cause the system entering an erroneous state. Such unexpected modifications of system state will lead to system failures hence need to be identified and corrected.

Sequence diagram C_2 illustrates the following scenario: during the execution of a requested operation, another unnecessary operation is activated or the same request is repeated by the component before the flow of control is transferred back to the application. As discussed previously, this may cause the changes to system state thus needs to be addressed. For example, if the requested operation of removing an employee's record (i.e. pseudo code remove(index)) were repeated, the system is in an erroneous state and subsequent operations may fail as a result.

In all two scenarios discussed above, even the operation activated unexpectedly does not have any effect on system state, its execution will request system resource (e.g. memory and CPU time). This may cause delays to normal system service provision. In a time-critical system this can still lead to system failure due to the inability to meet the strict timing constraints.

In scenario 3 (C_3 in Figure 3), one or more unexpected messages (e.g. m_{err}), among other legitimate messages (e.g. m_{send} or m_{read}), are sent/read by the component through a communication channel. This failure, in a broader sense, represents unintended interactions over an intentional communication channel. m_{err} can be the repetition of a previously sent/read message. Such a message, if undetected, received and acted upon by the application/component, could lead the system into an erroneous state and subsequent failures. Unexpected message sending may overwrite existing message (within the communication channel) that have not been read by the application. Unexpected message reading may result in an old message (already received and handled) being read by the component.

Detection and mitigation

A commission failure may have no effect on a system's normal operation at all, if the commission failure does not cause system state change and normal operations still finish within specified time boundary. This type of commission failures is benign and mitigation is unnecessary.

If a commission failure causes unexpected changes of system state, it can be regarded as a value failure and be addressed accordingly. If a commission failure causes deadlines of normal service provision being missed, it becomes a timing failure and should be treated accordingly. It is possible for a component commission failure to involve both value and timing anomalies. In this case, both value and timing effects will have to be detected and dealt with using appropriate techniques.

For the various cases of unexpected activation of an operation (e.g. unused functions), knowing their existence and causes are important. Exercised by a wide user base, an OTS component often has a list of known faults and corresponding inputs that can cause the manifestation of these faults. Known inputs that can lead to such failures may be avoided by using an input wrapper. Applying fault injection technique over the component's interface may offer some insights on detection of such failures. Monitoring usage of system resource and interactions between the component and the operational environment may also help detect such failures.

4.4. Omission failures

Representation, interpretation, and examples

An omission failure occurs when a requested service has not been provided. In a component-based system, such failures would appear in one of the two forms: a requested operation or received message has not been acted upon, and failure to send or read an expected message through a communication channel. These two forms of component omission failures are illustrated as three sequence diagrams in Figure 4.

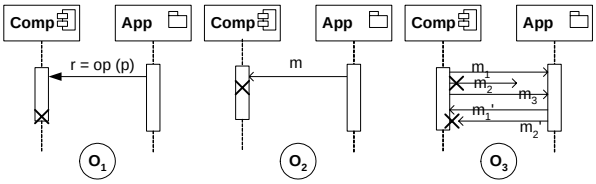

Figure 4. Omission type failure scenarios

In omission type failure scenario 1 (O_1 in Figure 4), upon sending a request for a procedure call, the application has not received the expected result or the flow of control has not been transferred back to the application within certain time limit. In omission type failure scenario 2 (O_2 in Figure 4), an asynchronous message sent to component has not been acted upon.

In omission type failure scenario 3 (O_3 in Figure 4), one or more periodic or aperiodic messages have not been sent or read by the component.

Detection and mitigation

An omission failure would inevitably result in expected system state changes or results not being delivered. Hence an omission failure is also a value failure. A deadline may also be missed as the result of an omission failure. For example, a procedure call fails to start or fails to finish within a specified time

interval. According to each individual omission failure scenario, appropriate techniques for handling timing failures or/and value failures can be selected to address the failure under consideration.

4.5. Summary

From the previous discussion (Sections 4.1 – 4.4), the applicability of different fault tolerance techniques in regard to the detection and mitigation of different types of failures of an untrustworthy component are summarised in Table 1.

This table merely provides some indication on the applicability of each fault tolerance technique for the detection and mitigation of different types of component failures. Individual cases still need to be examined in order to select the most appropriate fault tolerance techniques.

It is worth noting that, in a safety-critical application, choice of mitigation strategies must be determined by the nature of system hazards that can arise as the results of failures of an untrustworthy component. On detection of such failures, suitable mitigation strategies can be selected to make the system fail safe (if a well-defined safe state exists, e.g. shut down) or fail operational (where no safe state can be defined for the system, e.g., a flight control system, hence minimal level of service must be provided), or to make the faulty untrustworthy component fail silent (where erroneous services are more hazardous than no service at all, e.g. value and timing failures can be transformed into omission failures).

Table 1. Mitigation by failure type

(NOTE: N/A – not applicable, D – detection, M – Mitigation; T – Timing failure, V – Value failure, C – Commission, O - Omission)

Fault Tolerance Technique	Component Failure Type			
	T	V	C	O
Sanity checks	N/A	D	N/A	N/A
Reversal checks	N/A	D	N/A	N/A
Control + Monitor	D/M	D/M	D/M	D/M
Recovery blocks	D/M	D/M	D/M	D/M
NVP	D/M	D/M	D/M	D/M
Watchdog timer	D	N/A	D	D
Timestamp	D	N/A	N/A	N/A
Coding checks	N/A	D	N/A	N/A
Wrapper	N/A	N/A	N/A	N/A

Cost-effectiveness is also an important factor in selecting appropriate mitigation strategies: one fault tolerance technique may be able to detect and mitigate a group of identified component failures, and a combination of fault tolerance techniques may be used to detect and mitigate one critical failure whose impact onto the system safety is significant.

5. Conclusion

In order to use an untrustworthy component within a critical application with confidence, it is necessary for the system level effects of the component failures to be well understood and properly controlled. This can only be achieved through employing various fault tolerance techniques to deal with the specific component failures of concern. In this paper, we described the concept of mitigation by component failure types. By presenting a clear picture on various failure scenarios of a component, the approach facilitates the selection of cost-effective fault tolerance techniques for mitigating different types of component failures. With this approach, an untrustworthy component can be used within a critical application with increased confidence.

6. Reference

[1] Avizienis A., "The Methodology of N-Version Programming," in *Software Fault Tolerance*, R. Lyu, Ed.: John Wiley & Sons, 1995.

[2] Douglass B. P., *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*: Addison Wesley, 1999.

[3] Ezhilchelvan P. and Shrivastava S. K., "A Classification of Faults in Systems," University of Newcastle upon Tyne, Technical Report, 1990.

[4] Lee P. A. and Anderson T., *Fault Tolerance: Principles and Practice*, 2nd ed.: Springer-Verlag/Wien, 1990.

[5] Popov P., Strigini L., Riddle S., and Romanovsky A., "Protective Wrapping of OTS Components," *in Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, Toronto, Canada, 2001, IEEE Computer Society.

[6] Popov P., Strigini L., and Romanovsky A., "Diversity for Off-The-Shelf Components," *in Proceedings of the International Conference on Dependable Systems and Networks (DSN'2000, formerly FTCS-30 & DCCA-8)*, New York, USA, 2000, IEEE Computer Society.

[7] Pumfrey D. J., "The Principled Design of Computer System Safety Analyses," Department of Computer Science, University of York, PhD Thesis, 2000.

[8] Randell B., Lee P. A., and Treleaven P. C., "Reliability Issues in Computing System Design," *Computing Surveys*, vol. 10, pp. 220-232, 1978.

[9] Somani A. K. and Vaidya N. H., "Understanding Fault Tolerance and Reliability," *IEEE Computer*, pp. 45-46, April 1997.

[10] Torres-Pomales W., "Software Fault Tolerance: A Tutorial," Langley research Center, Hampton, Virginia, Technical Report NASA/TM-2000-210616, 2000.

[11] Villemeur A., *Reliability, Availability, Maintainability and Safety Assessment*, vol. 1 - Methods and Techniques: Wiley, 1992.