A Reliability-oriented Faults Taxonomy and a Recovery-oriented Methodological Approach for Systems Resilience

Carlo Vitucci
Technology Management
Ericsson AB
Stockholm, Sweden
carlo.vitucci@ericsson.com

Daniel Sundmark
Computer Science and Software Enigineering
Mälardalen University
Västerås, Sweden
daniel.sundmark@mdu.se

Marcus Jägemar
Sys Compute Dimensioning
Ericsson AB
Stockholm, Sweden
marcus.jagemar@ericsson.com

Jakob Danielsson
Sys Architecture
Ericsson AB
Stockholm, Sweden
jakob.danielsson@ericsson.com

Alf Larsson
Senior Specialist Observability
Ericsson AB
Stockholm, Sweden
alf.larsson@ericsson.com

Thomas Nolte
Division of Networked and Embedded System
Mälardalen University
Västerås, Sweden
thomas.nolte@mdu.se

***Abstract*—Fault management is an important function that impacts the design of any digital system, from the simple kiosk in a shop to a complex 6G network. It is common to classify fault conditions into different taxonomies using terms like *fault* or *error*. Fault taxonomies are often suitable for managing fault detection, fault reporting, and fault localization but often neglect to support all different functions required by a fault management process. A correctly implemented fault management process must be able to distinguish between defects and faults, decide upon appropriate actions to recover the system to an ideal state, and avoid an error condition. Fault management is a multi-disciplinary process where recovery actions are deployed promptly by combined hardware, firmware, and software orchestration. The importance of fault management processes significantly increases with modern nanometer technologies, which suffer the risk of so-called soft errors, a corruption of a bit cells that can happen due to spurious disturbance, like cosmic radiation. Modern fault management implementations must support recovery actions for soft errors to ensure a steady system. This paper describes an extended fault classification model that emphasizes fault management and recovery actions. We aim to show how the reliability-based fault taxonomy definition is more suitable for the overall fault management process.**

***Index Terms*—Reliability, Fault management, Fault topology, Fault taxonomy;**

I. Introduction

Today's systems are becoming more complex due to the higher number of supported functions. As a result, faults in the system can cause system failure, which can be fatal for large infrastructure systems. We propose using fault management to ensure that a faulty system can continue running with limited but correct functionality until maintenance action can replace the suspected faulty component according to a planned maintenance phase. Dori et al. [1] defines a system as: "...an integrated set of elements, subsystems, and assemblies that accomplish a defined objective. These include products (hardware, software, firmware), processes, people, information, techniques, facilities, services, and other support elements". The above definition emphasizes that basic but essential functions are integral to the system design regardless of the service complexity or components. Fault management is one of the crucial differentiating functions. It is an always-present system function spanning from a kiosk in a shop [2] to complex end-to-end vertical service deployment [3]. It could not be otherwise because the most critical value of each product is the ability to comply with the requirements. Therefore, particular attention is given to Quality of (user) Experience (QoE) [4] and fault management is how a system detects and reacts to defects and continues to work according to its specification [5], [6]. For that reason, this paper considers fault management as a critical service to achieve QoE and reduce maintenance costs.

The *key contribution* of this paper is:

- A novel fault taxonomy that considers the product reliability.
- A revised fault management architecture including the fault isolation concept for limiting the fault propagation into the system extending the model used by Wuhib et al. [7].

Systems reliability has been a growing field of research in recent years. There seem to be two directions most explored: increasing hardware fault tolerance [8], [9], and the definition of a better taxonomy [10], [11]. Both fault tolerance and taxonomy research look at the single hardware component, at embedded system-level [12], and at the architecture of distributed systems [13].

Our proposal joins the existing works and moves attention towards a non-frequently explored direction: fault recovering.

978-1-6654-8810-5/22/$31.00 ©2022 IEEE
DOI 10.1109/COMPSAC54236.2022.00016

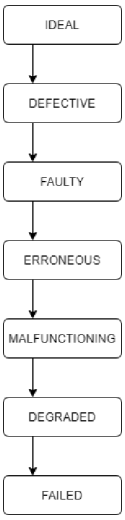

Fig. 1. Transition state of a reliability multilevel model.

This paper is structured as follows. First, we start by introducing the fault management terminology, architecture, and the significant characteristics of the discipline in section II. Then, after summarizing a standard reference fault typology in section III, we show an initial revised version in subsection III-A and a complete reliability-oriented classification in subsection IV-A. Thanks to this new model, we describe a typology between fault classification and recovery actions in section V. Eventually, section VI shows a use case as an example of the introduced typology.

II. Fault management architecture

Faults are often incorrectly mixed with other negative system states such as failures and errors. We use Parhami's [14] definition of a system's different working states, listed as follows;

- **Ideal**: The state where the system is performing according to its specification.
- **Defective**: The state where an incorrect behavior is detected.
- **Faulty**: The state where a behavior outside specification is manifested in the system, causing an error.
- **Erroneous**: The state where the system may start working with behavioral deviation from the ideal state if it cannot execute the needed recovery actions.
- **malfunction**: The state where the system is working with behavioral deviation from the ideal state.
- **degraded**: The state of a system able to support a malfunction behavior thanks to tolerance provisions.
- **Failed**: The state where a system is not able to meet the specification.

The transition state model described in Fig. 1 shows the possible system states. It is worth mentioning how the description of the transition model state emphasizes the system's ability to avoid a failed condition using both recovery actions and

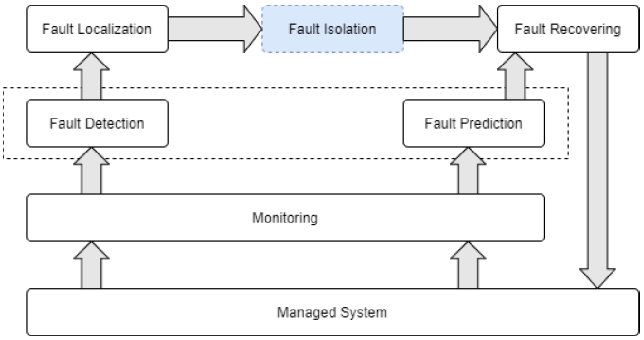

Fig. 2. The fault management system.

tolerance provisions. Defective is the state where hardware or software may deviate from the ideal system state. Wear and aging of the hardware may cause a defect and may, in turn, bring the system into a faulty state. A faulty condition may or not cause an erroneous state if no recovery action is possible. An erroneous state may or not cause the affected system to malfunction depending on the system's error tolerance. A malfunction does not necessarily mean a perceivable service-level drawback, but if that happens, the degradation of the service has the consequence of a failed system condition.

The transition state model must be the reference point of every well-designed fault management process since fault management aims to verify that the system is working according to its functional requirements.

The fault management function is more complex than a primary error detection mechanism. The goal of the fault management function is to guarantee that the system is working within a specific performance range and at the full availability of provided service. The customer perception and the product maintenance cost are the true meaning of a guaranteed QoE. Ensuring QoE requires a prompt reaction to the system disturbances that may compromise the quality of the service for the end-user. Fault management cannot be only the supervision of possible faults but must implement the four pillars of the recovery attempt [15] in mind. We list these pillars as follows:

- **Fault detection**. Automatic collection of alarms as generated by hardware devices or system software components. The fault detection phase may be a simple check of any possible fault. A more complex system may be a full monitoring mechanism in a vast range able to collect data and metrics. The fault management can also be based on artificial intelligence/machine learning to be able to predict faults [7].
- **Fault localization**. Alarm data and metrics preparation, increasing the value of the information provided by the fault detection. The scope of the fault localization is to provide a useful indication of the failure's root cause. For that reason, the representation of time and temporal relation among events [16] is an important fault localization aspect.
- **Fault isolation**. The fault isolation scope is to reduce the

fault propagation condition. It resolves the relationship between different events, creating a priority-based alarm classification to identify root cause interfering conditions. Usually, it is based on hardware and software features to mark a faulty state in an active alarm condition. One example is the usage of temporal and memory isolation as well as the poisoning algorithm to mark memory as corrupted [17].

- **Fault recovering**. Application of specific recovery or dispatch actions to remove the actual root cause of the fault indication (fixing the fault). It can also be to reshuffle resource availability isolating and removing the faulty component to bring back the system to an operational state [18].

We consider the fault management described by Wuhib et al. [7] as the baseline for our study. In Fig. 2, we extend the baseline model by adding a fault isolation component. We argue that the added element is essential for the fault recovering process since it allows a better identification and recovery of the faulty device. Moreover, it is helpful to design fault management as follows:

- **A Multi technologies area**: Fault management should not solely interest the software hardware-fault supervision handler. Instead, efficient fault management combines hardware, software, and system into a single holistic fault domain.
- **Applicable to multiple development scenarios**: Providers increasingly deploy services as scalable and modular in distributed systems to meet user expectations. A fault management design that does not consider this service deployment trend and system characteristic would have a severely limited domain of validity.
- **Product life cycle oriented**: The product quality attitude starts from the very early stages of the product's life. Fault management design makes the difference if it adds value along the entire products' life cycle, from production to repair centers, from function test to end-of-life.
- **Based on multiple tools and sensors**: Fault management uses all possible techniques and features to improve efficiency. Hardware Supervision, together with hardware tests, diagnostics, behavioral observability, and, at least at some levels, security, shall be considered tools and sensors of fault management, so components that must be used at the right time to perform the proper action.

III. Error typology

Fig. 3 depicts an established fault classification scheme [19]. This classification identifies the fault by component, location, and function integrity. It is a practical example of a fault taxonomy driven by fault supervision. Still, it doesn't provide any valuable information for the fault recovering state of the fault management architecture shown in Fig. 2.

A. Revised taxonomy model

In this section, we introduce a revised model for the fault taxonomy. Our model proposition introduces the priority-based

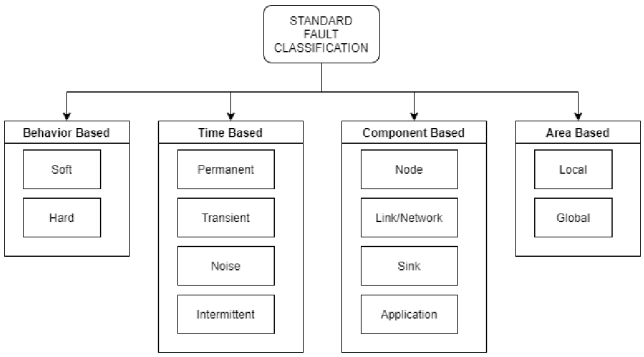

Fig. 3. Standard Fault Classification.

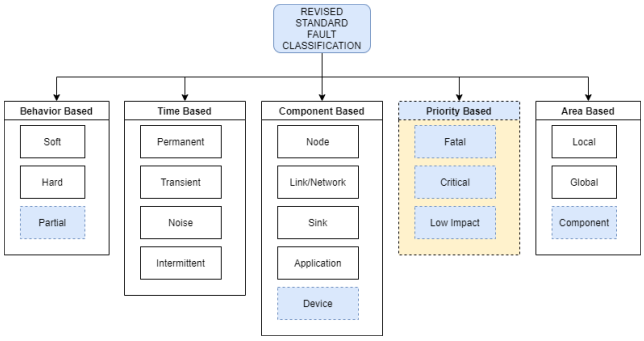

Fig. 4. Revised Fault Classification.

class (fatal, critical, and low impact fields). In addition, a "Partial" field enriches the Behavior-based class, a "Device" field extends the component-based class, and a "Component" field enhances the area-based class. The new fields aim to increase the product's reliability in its lifetime through improved fault information for the fault recovering mechanism. Fig. 4 depicts our proposal.

We describe the newly introduced fields in detail as follows:

- **Behaviour-based**: The behavior-based class in Fig. 3 shows "soft" and "hard" fields. They refer to the ability of the managed system to remain active. A "hard" fault identifies general system inactivity "hard" fault. In contrast, a "soft" fault addresses a system that remains in contact with the other network elements in an unreliable working condition. We add the "partial" classification to be able to manage more cases in a fully reliable system: a system that can still partly perform its task but with lower performance. The difference between soft and partial error is in their definition: the first one addresses the capability of the system to remain active, the second to work under reduced performance. It could be a performance degradation caused by an aging issue or the consequence of a recovering action for a malfunction state that removes the faulty object from the list of the available resources to let the system work, even if with limited specifications. It is a decision taken considering that, for the customers, a system with lower performance,

matching its specification, is more valuable than a system with perhaps higher performance, but in non-reliability conditions. For example, it is well known [20], [21] that the customer usage of a network is lower than the maximum system capability for most of the day. Then, a dynamic reallocation of resource availability is possible: the system can rethink resource usage after isolating the faulty component in a manner that can still grant reliable lower performance and untouched QoE in non-rush hours.

- **Component-based**: the ”device” fault characterization introduced in the taxonomy is considering the ”fault localization” phase. The possibility of having a more exemplary fault indication is necessary to manage better the redistribution of the hardware resources left available in case of a partial error condition handling.
- **priority-based**: this classification is a new extension to [19] and its purpose is to manage the ”fault isolation” phase. Correct root cause analysis must consider fault priority to distinguish between primary and secondary faults. The scope of fault management is to identify the actual root cause of a possible faulty state. Until a primary fault condition is active, fault management must postpone any secondary fault analysis. *Fatal* is the highest primary for a fault because it sets the system in an unstable condition. Fault management may ignore any other type of fault indication. *Critical* is a primary fault. Fault propagation is possible because a primary fault may compromise the working condition of other system functionalities. *Low impact* is a secondary fault. The likelihood of fault propagation is very low. Fault management must analyze other concurrent fault conditions too. For example, systems that operate under extreme temperatures are not reliable [22]: the automatic temperature control policy may reduce the CPU frequency, and the clock may lose its stability. Fault management must consider the high-temperature condition as the highest critical fault and ignore all secondary fault indications. Fault Management will evaluate CPU frequency and clock instability issues if it detects those conditions after the High-temperature recovering action.
- **Area-based**: ”Component”, ”Device” and ”Partial” fields allow a better fault indication resolution. For example, the fault of a particular memory array can be defined as ”partial” because the system can continue to work isolating the corrupt memory array. The ”Device” field will be equal to the ID of the memory with the problem, and ”Area-based” will be ”component” because it is limited to memory only, without any propagation of the error to other hardware components and without considering the fault as at the board or node level.

IV. Reliability consideration

The revised version of the standard fault taxonomy is incomplete. The actual reason for adding the new fields is to improve error handling. Still, we can resume the discussion by reconsidering the state transition model in Fig. 1 to understand what kind of classification is still missing. The complete scheme indicates that a verification state is associated with each node: the method evaluates the actions necessary to restore the system in reliability conditions for each state. Similarly, external actions can result in a change of state, and Fig. 5 shows the complete scheme of the transition state model. The full version of the transition model indicates that any interstate movement is reversible. Furthermore, it shows that it is required to find the best solution to deal with a fault condition.

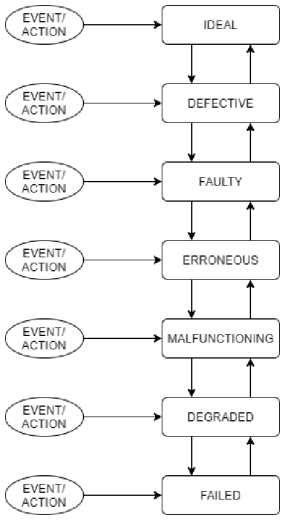

Fig. 5. Complete transition state of a reliability multilevel model.

A. Reliability Fault classification

We add another class to the standard fault classification to describe how a system’s reliability may change due to new faulty conditions. The most helpful information about a fault is which type of action is possible to recover from a non-ideal state or if the system can react to an external event that can change the state of the transaction state model. We distinguish three different types of faults: ”Corrected”, ”Correctable”, and ”Uncorrectable”, described in detail as follows.

1) Corrected Fault: The system has already corrected the fault from a self-recovering action taken by hardware, firmware, or software. The system will remain in its current reliability state after the recovery action. The corrected error events are helpful for the fault prediction task, which aids the system in making specific recovery actions based on the fault prediction analysis results. For example, an excessive number of corrected fault events can be an indication of system aging, implying that specific hardware components must be replaced [23].

2) Correctable Fault: The system marks an actual fault as ”correctable” when it knows how to counteract the fault to avoid drawbacks to both functions and performance. During ”correctable” faults, the system is in the ”defective state” of the reliability transition model. Therefore, it must fix the fault to void that consecutive ”correctable faults” bring the system

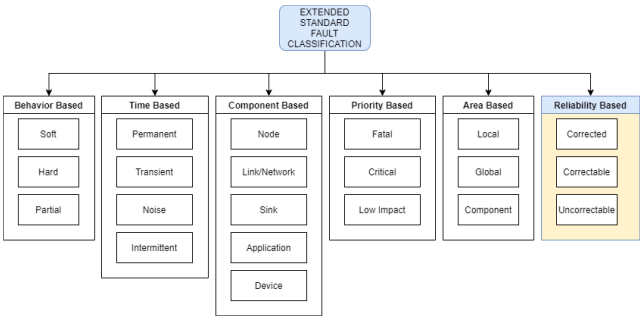

Fig. 6. Complete transition state of a reliability multilevel model.

into an ”uncorrectable fault” state. A typical example of a correctable fault is the memory Error Correction Code (ECC) algorithm. When ECC reports a fault as ”Correctable”, the hardware or the software will write the corrected value to the memory address [24].

3) Uncorrectable Fault: The fault handler doesn't know how to recover from the faulty state, and the faulty component becomes unreliable. It could be a board device or a more severe machine check exception for the processor. A typical uncorrectable error is when the ECC algorithm detects a double error in the memory, and it is not able to report the correct data value anymore [25]. It is worth mentioning that the faulty state is not the final stage of the transition state model. The system can consistently execute proper actions to recover to the ideal working condition avoiding the erroneous state, not only in case of ”corrected” or ”correctable” indications. The discriminating factor for recovering plans is the system's unreliability: any possible action may request a restart of the board to have a more stable condition. In the best case, a re-initialization of the faulty component can recover from the fault condition. However, it is challenging to avoid temporary disturbances to the end-user service. The system migrates from a faulty to an erroneous state only when all possible recovery actions fail. For example, the system may start using only the available resources, isolating the faulty objects. Consequently, the limited availability of resources can partially meet the specification: the system marks the fault as partially recovered and will work in a degraded state. The failed condition is only the latest stage when even a partial recovery from the system is impossible.

We propose our final classification scheme in Fig. 6 that contains all states from the standard classification scheme and the addition of our two new classes (priority-based and reliability-based).

V. The Reliability Topology

Our extended classification scheme allows for a better understanding of the priority of detected errors. The classification is no longer exclusively suitable for information classification for fault supervision but now also enables us to identify the recovery action for a system that is not in an ideal state.

The key passage of this new fault classification is the orientation towards system recovery. It is a new approach for designing the fault management framework that arises from the awareness of the high costs of maintenance. Reporting a fault without trying to remedy its presence produces high costs, heavy impacts on the service, and a lousy product reputation in customers' perception. The successive step is to identify a topology between the reliability class and the specific action sequence that can recover from a faulty state because the value of the reliability class drives the actions to solve a fault case.

Corrected faults are already resolved and should, therefore, never interrupt the system to avoid performance drawbacks. ”Corrected” faults require no action other than the supervision of their occurrence to be able to detect and report a suspicious frequency. Fault prediction algorithms can use ”Corrected” fault statistics too.

Correctable faults require action. The fault indication is propagated in the system with an interrupt so that the system reacts immediately to its presence and carries out the recovery action.

Uncorrected faults are managed through exceptions and require a restart of the faulty component as a recovery action because the system has become unreliable.

Recovery actions include simple mechanisms such as re-initializing objects (e.g., memory write-backs) or more complex mechanisms such as re-initializing both hardware and software, which is significantly more time-consuming. Sometimes, the re-initialization of a faulty component requires a board restart. In other cases, the isolation of a faulty component requests a reconfiguration of the system. One more time, It is worth remembering that the system must always try recovery action, not exclusively for correctable errors. Uncorrectable fault or a high-occurrence frequency of corrected fault may also request a recovery action sequence. We categorize the behavior of faults into three states; soft, partial, and hard to determine a system's reliability level. A fault is marked as soft when a recovery action can move the system back into an ideal state of the transition model. A partial state means that the recovery action identifies a new system resource configuration isolating the uncorrectable fault and a degraded function condition. ”Hard fault” indicates that an uncorrectable fault is critical for the system because it can be neither recovered nor isolated. As a result, the fault compromises the system's overall functionality, and the erroneous state is unavoidable.

Fig. 7 shows the topology between the reliability class and the recovery actions.

Our mechanism gathers a set of information for every detected fault (see object classification in Fig. 7), which is essential to execute all stages of the fault management framework: fault detection, localization, isolation, and recovery. Fault management events may be stored and retained for posthumous analysis. ”Report” propagates the set of fault information and ”log” stores them. Usually, the term ”log” indicates the file constructed with the data associated with the fault event. However, to avoid possible confusion, we

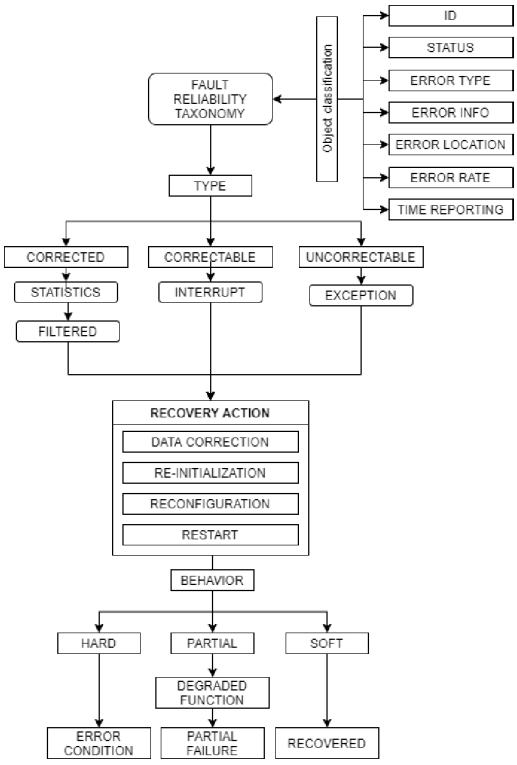

Fig. 7. Recovering actions and reliability class topology

distinguish between ”log” and ”report”: both can contain the same information, but the ”report” is the file propagating the information relating to a fault in the system, while the ”log” is a file that contains error information when it occurs and it stores, not propagates, the fault info in the system, except for statistical analysis. Therefore, the ”report” must contain the information necessary for the elements of superior control to better understand the event and the best action with which to handle it. The report propagates the set of information in Tab. I

TABLE I
REPORT INFORMATION, FOR COMPONENT IN A FAULTY STATE

object parameter	Value	Comment
ID		unique object identity tag
Status	No Fault Faulty Defective Degraded	Used to cease a previosuly triggered faulty/defective case
Reliability	Correctable uncorrectable corrected	An action may recover the error condition Object is not trustable, it cannot continue to work fault has been identified, isolated, and corrected
device info	Ethernet port Equipment clock temperature sensor memory device etc.	
location	node board device offset absolute memory address	where the fault is located
Error rate		an implicit indication that a fault must be detected based on a filtering mechanism, through occurrences or consecutive error occurrences within a time-based windows
Timing		a global timestamp info, showing when the fault event is set, or fault entry is added to a report (or log)
Recovery action	Re-initialization of the object Restart of the board Restart of the board and board test Change	which type of action is needed to recover from the faulty/defective state Repair action is needed
Recovery time		Time according to the general recovery escalation principle, i.e., the time within a new occurrence of the same fault escalates to the next step

VI. HARDWARE SOFT ERROR IN MODERN COMPUTING SYSTEM

The usage of technology in the nanometer range has made the quality of the product even more critical [26]. For example, it is well-known [27], [6] that working with the nanometer's technology increases the probability of the so-called hardware soft errors for the SRAM memory devices, and the soft errors may also be a relevant problem for the DRAM and logic circuit. Therefore, the soft error is expected, and the recommendation is not only to detect it but also to implement the recovery actions, from design (preventing malfunction) to real-time (writing back the correct value). It is no longer enough only to supervise a fault; the complexity of new technological solutions obliges us to remain compliant to fault management target: *detecting, isolating, and correcting* any fault indication before it becomes an irrecoverable error. Tab. II shows the recommendation for hardware soft error handling. The recommendation addresses fault occurrence reduction in the design phase and fault isolation and recovery strategy in the run-time phase.

A. Use case: Error Correction Code (ECC)

Error correction code (ECC) is a mathematical process widely used to protect the data stored in memory from corruption. It uses a single bit to detect errors in a more significant number of bits and provides an algorithm to recreate corrected data in real-time. Since most computer systems move data in chunks of 64 bits, the ECC algorithm generates seven extra bits per 64 bits of data. In case of a single-bit error, the ECC algorithm can notify the correct value and the event to the system. Several ECC technologies are available, and the hardware takes recovering actions depending on the implemented process. E.g., the single-error correction and

TABLE II
PRINCIPLES FOR SOFT ERROR MEASURE [28]

Principles	Mitigation techniques	Action
Reduction	1	Change in materials
	2	Work on physical structure
	3	Reduction in areas where soft error occur
Isolation	1	Work on circuit configuration
	2	Identification of parts with and without substantial function
Correction	1	Automatic correction in hardware
	2	Automatic correction in the equipment control program
	3	Correction in accordance with maintenance personnel operation

double error detection (SECDED) or the Hamming code that are extremely popular ECC mechanisms [29]. Since ECC can detect and correct a corrupted bit, it is an excellent example because hardware uses the reliability class to report bit error detection to the system depending on the implemented ECC algorithm. Hardware, firmware, and software can take different recovering actions based on the value of the reliability class.

Tab. III shows the application of the topology to the case of ECC memory fault. The ECC use case helps to understand the concept behind our research: each system component and technology reacts continuously to a fault indication. The ultimate goal is not "fault detection" but the "fault recovering". Hardware, software, and higher management components analyze the fault information as detected, take an active part in the correction of the fault, and participate in the processing of the fault during its propagation in the system. Our solution emphasizes the weight of the choral action of fault recovering. Without multi-level orchestration between hardware and software for fault information processing, a system can only detect the error. The absence of a continuous fault recovery attempt compromises the product service more or less seriously. Moreover, it generates a wrong perception of the product for the end-user.

VII. CONCLUSIONS AND FUTURE WORK

The quality of the service is a discriminating element of any product and can cause an unsatisfactory end-user QoE. Investing in fault management where reliability comes to have a dominant position also means saving a lot of resources for the maintenance task. Maintenance task means intervention by qualified personnel in site, repair centers activities, and troubleshooting. It is well-known to be of a significantly higher magnitude than any reliability design cost in the early life of the product. In a future task, we want to analyze the cost of the fault and the cost of fault management in a more structured manner. We want to verify if the same fault may have a different cost if fixed by a well-designed fault management strategy or maintenance

TABLE III
RELIABILITY TOPOLOGY FOR ECC USE CASE

ECC fault type	discipline	recovery action
corrected	HW	no action, hardware already corrected the single bit corruption number of corrected error are available for statistics
Correctable	HW	corrected value reported to the software
	FW	write back to the physical memory
	SW	patrol or manual scrubbing: time based background walk through the whole memory to correct detected fault, avoiding a second bit corruption in the same row to cause an uncorrectable fault.
Uncorrectable	HW	Two or more errors detected. ECC process is not able to fix the problem
	SW	Memory pollution. Mark the memory as corrupted, avoid the fault propagation to the software, in case the corrupted memory is going to be used
	SW	Memory initialization. Most of the time, a soft error is recovered by means of memory reinitialization
	SW	warm restart, when a reinitialization of the memory cannot be done without shutting-down the applications
Uncorrectable, persistent	HW+FW+SW	walk through consecutive warm restart to detect if fault has been fixed
	SW	Persistent fault row could be removed from the list of available memory (degraded function)

activity. The integration between supervision-oriented and reliability-oriented taxonomy enables a new fault handling approach where fault recovery is central. Thanks to the new taxonomy, it is simpler to provide a topology that maps a fault indication into the domain of the action taken to recover from a faulty state before it becomes an erroneous (or, even worse, a failed) state. This topology will be the starting point for a future reworking of the fault management framework. We think the new framework will also establish a series of hardware and software requirements for next-generation products.

REFERENCES

[1] D. Dori, H. Sillitto, R. M. Griego, D. McKinney, E. P. Arnold, P. Godfrey, J. Martin, S. Jackson, and D. Krob, "System definition, system worldviews, and systemness characteristics," *IEEE Systems Journal*, vol. 14, pp. 1538–1548, 6 2020.

[2] Eazy-Q, "Queue management system — queuing — que solution —." [Online]. Available: https://www.smartmatrixuae.com/service-cat/queue-management-solution/

[3] ETSI, "Ts 128 545 - v16.1.0 - 5g; management and orchestration; fault supervision (fs) (3gpp ts 28.545 version 16.1.0 release 16)," 2020. [Online]. Available: https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx

[4] P. Brooks and B. Hestnes, "User measures of quality of experience: why being objective and quantitative is important," *IEEE Network*, vol. 24, no. 2, pp. 8–13, 2010.

[5] I. Chih-Lin, S. Kukliński, T. Chen, and L. L. Ladid, "A perspective of o-ran integration with mec, son, and network slicing in the 5g era," *IEEE Network*, vol. 34, pp. 3–4, 11 2020.

[6] S. Cherrared, S. Imadali, E. Fabre, G. Gossler, and I. G. B. Yahia, "A survey of fault management in network virtualization environments: Challenges and solutions," *IEEE Transactions on Network and Service Management*, vol. 16, pp. 1537–1551, 12 2019.

[7] F. Wuhib, C. Fu, and M. Soualhia, "A look at automated fault management with machine learning - ericsson," 2019. [Online]. Available: https://www.ericsson.com/en/blog/2019/6/automated-fault-management-machine-learning

[8] Y. Chen, G. Lin, E. Crowe, and J. Granderson, "Development of a unified taxonomy for hvac system faults," *Energies*, vol. 14, p. 5581, 9 2021.

[9] M. Smara, M. Aliouat, S. Harous, and A.-S. K. Pathan, "Robustness improvement of component-based cloud computing systems," *The Journal of Supercomputing*, 9 2021.

[10] S. Bruning, S. Weissleder, and M. Malek, "A fault taxonomy for service-oriented architecture." IEEE, 11 2007, pp. 367–368.

[11] G. P. Bhandari and R. Gupta, "Extended fault taxonomy of soa-based systems," *Journal of Computing and Information Technology*, vol. 25, pp. 237–257, 1 2018.

[12] S. Shu, Y. Wang, and Y. Wang, "An approach to architecture-based fault tolerance evaluation with fault propagation," *Proceedings of 2015 the 1st International Conference on Reliability Systems Engineering, ICRSE 2015*, 12 2015.

[13] M. Barranco, S. Derasevic, and J. Proenza, "An architecture for highly reliable fault-tolerant adaptive distributed embedded systems," *Computer*, vol. 53, pp. 38–46, 3 2020.

[14] B. Parhami, "Defect, fault, error, ..., or failure?" *IEEE Transactions on Reliability*, vol. 46, pp. 450–451, 1997.

[15] M. Nouioua, P. Fournier-Viger, G. He, F. Nouioua, and Z. Min, "A survey of machine learning for network fault management," pp. 1–27, 2021.

[16] M. Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," *Science of Computer Programming*, vol. 53, pp. 165–194, 2004.

[17] A. Kleen, "hwpoison — the linux kernel documentation," 2009. [Online]. Available: https://www.kernel.org/doc/html/latest/vm/hwpoison.html

[18] A. Menychtas and K. G. Konstanteli, "Fault detection and recovery mechanisms and techniques for service oriented infrastructures," pp. 259–274, 10 2011.

[19] E. Moridi, M. Haghparast, M. Hosseinzadeh, and S. J. Jassbi, "Fault management frameworks in wireless sensor networks: A survey," pp. 205–226, 4 2020.

[20] H. D. Trinh, L. Giupponi, and P. Dini, "Urban anomaly detection by processing mobile traffic traces with lstm neural networks," *Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks workshops*, vol. 2019-June, 6 2019.

[21] J. L. Bejarano-Luque, M. Toril, M. Fernandez-Navarro, C. Gijon, and S. Luna-Ramirez, "A deep-learning model for estimating the impact of social events on traffic demand on a cell basis," *IEEE Access*, vol. 9, pp. 71 673–71 686, 2021.

[22] T. Instruments, "The engineer's guide to temperature sensing," vol. 2019.

[23] T. Liu, C. C. Chen, W. Kim, and L. Milor, "Comprehensive reliability and aging analysis on srams within microprocessor systems," *Microelectronics Reliability*, vol. 55, pp. 1290–1296, 8 2015.

[24] V. Sridharan, N. D. Bardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems the good, the bad, and the ugly," *ACM SIGPLAN Notices*, vol. 50, pp. 297–310, 4 2015.

[25] A. Lanzaro, A. Pecchia, M. Cinque, D. Cotroneo, R. Barbosa, and N. Silva, "A preliminary fault injection framework for evaluating multicore systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7613 LNCS, pp. 106–116, 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-33675-1_9

[26] K. Kang, H. Kufluoglu, K. Roy, and M. A. Alam, "Impact of negative-bias temperature instability in nanoscale sram array: Modeling and analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 1770–1781, 10 2007.

[27] I.-T. K.139, "Reliability requirements for telecommunication systems affected by particle radiation," 2018. [Online]. Available: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=13718&lang=en

[28] I.-T. K.131, "Design methodologies for telecommunication systems applying soft error measures," 2018. [Online]. Available: https://www.itu.int/ITU-T/recommendations/rec.aspx?id=13454&lang=en

[29] S. Mukherjee, "Error coding techniques," *Architecture Design for Soft Errors*, pp. 161–206, 1 2008.