System Self Diagnosis for Industrial Devices

Markus Rentschler, Stephan Kehrer
Hirschmann Automation & Control GmbH
Stuttgarter Straße 45-51
72654 Neckartenzlingen, Germany
{Markus.Rentschler,
Stephan.Kehrer}@belden.com

Clemens Pirmin Zangl
Hochschule Karlsruhe
Moltkestraße 30
76133 Karlsruhe, Germany
zacl1011@hs-karlsruhe.de

Abstract

Downtimes of failed devices in an industrial plant must be kept to a minimum to achieve high system availability. Since failures are often caused by transient hardware- or software faults, a well-defined System Self Diagnosis (SSD) functionality is an important feature for the effective long-term operation of industrial plants. To achieve effective SSD techniques for network infrastructure devices, a root cause analysis of past failures was conducted and a fault model derived. A set of error detection methods was derived based on the checked root cause indicators. For productive deployment, an extensible SSD framework in form of a rule based system was designed and implemented to be used as an embedded software tool throughout the whole product lifecycle of an industrial device, achieving a highly efficient "Design for Testability" approach.

1. Introduction

Industrial plants are often operated on the basis of an Industrial Ethernet network [1], over which the plant devices communicate. These installations are usually composed of various infrastructure devices such as switches and routers, interconnected within complex wired and wireless topologies (Figure 1). The plant operation relies on the reliable operation of the data communication via this network.

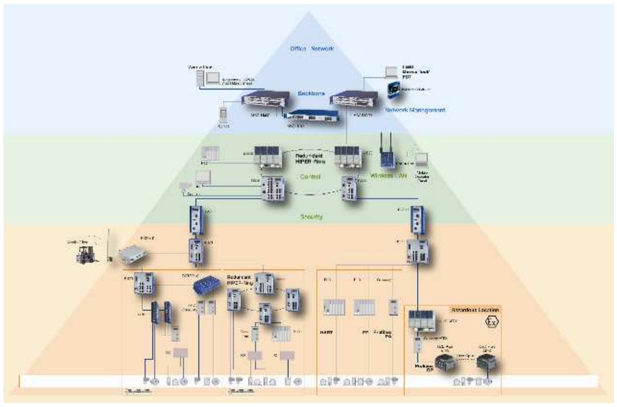


Figure 1. Industrial Ethernet Network in the Automation Pyramid

After an industrial network has been setup and is operating productively, downtimes have to be avoided. In the case of unexpected failure of network components, their recovery or replacement has to be performed in such a way as to keep the downtime of the affected parts of the network to an absolute minimum [2]. In case of permanent hardware failures, the device has to be physically replaced, which is a manual task performed by maintenance personnel. In the case of software failures, physical replacement is not required but rather appropriate recovery measures have to be undertaken to bring the device back to its correct operational state. Compared to the detection of faulty hardware components, which already is a non-trivial problem, the reliable detection of software failures in systems is even more challenging. Suitable error detection and associated recovery mechanisms are therefore required components for dependable self-healing systems.

The contribution of this work is a "Design for Testability" approach [3] through the definition of error detection rules and associated recovery strategies for embedded software based on the operating systems Linux and VxWorks [4]. A process model as basis for generic diagnostic rules identification is also summed up in this paper. Finally, a generic rule-based SSD framework is proposed.

The structure of the paper is as follows: in section 2, we sum up the established work on dependability and define the scope of our work. In section 3 a failure model for our type of system is defined. In section 4 an architectural analysis of a typical network device is performed to derive a system model. This model is then used to identify the critical system components by performing a root cause analysis on historical bug tracking data. Section 5 presents general error detection mechanisms. These are then used in section 6 as a base for developing the error detection methods and algorithms used in our approach. In section 7 we briefly discuss some approaches on system failure recovery. Section 8 presents our solution of a SSD framework. Finally in section 9 we summarize our work.

2. Dependability

Dependability techniques are a much researched subject. Numerous publications provide a good coverage of its various aspects which are most importantly availability, reliability and maintainability [5]. Dependability is largely driven by spacecraft, aircraft and automotive industry for the development of long mission and safety critical systems. Initially focusing on hardware designs, the same principles were later also applied to software based systems and further developed in that direction. In [6], a good overview on the subject and its history is provided.

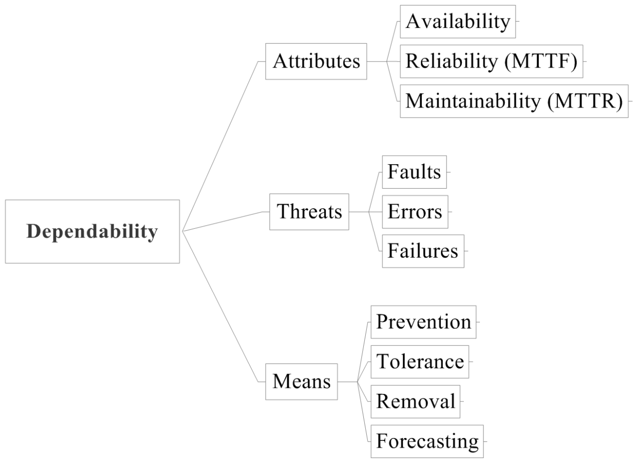


Figure 2. Dependability Taxonomy [5]

We consider the most important dependability attribute for industrial automation systems to be availability, which is basically a function of reliability and maintainability (Figure 3).

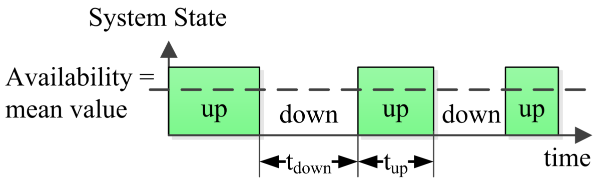


Figure 3. Availability Model

Availability A is the degree to which a system is within a defined and operable state and depends on *MTTF* and *MTTR*.

Mean Time to Failure (MTTF) expresses the reliability of a system by the mean time a system is expected to process correctly.

Mean Time to Recovery (MTTR) expresses the maintainability of a system by the mean time required to bring a failed system back into operation.

$$A = \lim_{t \to \infty} \frac{\sum t_{up}}{\sum (t_{up} + t_{down})} = \frac{MTTF}{MTTF + MTTR} \qquad (1)$$

As can be seen in (1), availability improvement can be achieved with a higher reliability (MTTF) and/or a better recoverability (MTTR). Although a high MTTF does not guarantee failure-free operation, a lower MTTR always minimizes the impact of failure. Thus lowering MTTR is obviously a powerful strategy to improve availability.

2.1. Dependability Threats

"A system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service" [5]. A failure is the manifestation of an error caused by a system fault. If the system is comprised of various interacting components, the failure of one component might introduce a fault in another component (Figure 4).

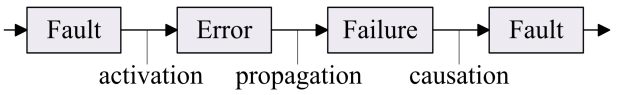


Figure 4. Chain of dependability threats [5]

2.2. Dependability Means

According to Laprie [5] et. al., the following basic methods for creating dependable systems shall be distinguished in the context of this work:

Fault Prevention (or fault avoidance) removes the source of faults (i.e. programming or design flaws) in the design phase prior to operation of the system in order to prevent faults from happening. This improves system reliability which is indicated by the mean time to failure (MTTF) thus directly increasing availability. Fault avoidance can be achieved through quality-focused development methods. Model-based code generation [5], test-driven development or design rules are typical fault-prevention measures.

Fault Removal applies validation and testing techniques to detect and remove faults before a system is put into operation. It can also be applied during system runtime by recording failures and removing them by updating the software during a maintenance cycle. This approach improves availability by making the system more reliable (higher MTTF). Code analysis, static and dynamic testing or advanced testing techniques such as software fault injection (SFI) [7,8] are typical measures. According to Dijkstra however "Program testing can be used to show the presence of bugs, but never to show their absence!"[10].

Fault Tolerance assumes faults to be present during system operation. It employs design techniques which ensure the continued correct system operation. One method is improving system availability through reliability (MTTF). Another method is the recovery of the system, thus also improving the availability by reducing downtime (MTTR). Fault tolerance can be achieved by utilizing design redundancy [10] but also by using error detection and recovery strategies.

Fault Forecasting uses techniques to predict commonly occurring faults to avoid or remove them or circumvent their effects. This approach improves availability through improvement of reliability (MTTF). Fault forecasting can be achieved by utilizing monitoring and prognosis.

2.3. Scope

The functionality of industrial automation equipment is often based on complex software systems, which cannot be developed completely error-free. This is due to the fact that a combinatorial explosion of possible system states takes place which cannot be all verified by testing. Thus, there is always some probability of faults remaining in the software. These faults can transiently and non-deterministically lead to error and subsequent failure. As a consequence they can cause productivity loss within the plant. Since availability must be at a maximum at all times, this probability of non-deterministic software failures has to be dealt with in some manner.

A failure root cause analysis in an existing bug-tracking database for complex embedded network products will be conducted to have a solid analytical base of the most important faults that result in expensive failures which are hard to debug.

In the process of finding a solution, we will focus on the aspects of fault removal and fault tolerance in the form of single-processor embedded SSD techniques that increase system availability through reduction of MTTR. This kind of fault detection followed by isolation and system reconfiguration is often referred to as "FDIR" process that can also be performed online [11][12]. We will focus in this work however on the diagnostics part and only shortly outline possible approaches for recovery.

3. Failure Modeling

3.1. Fault classification

Permanent faults are hardware or software design faults whose resulting failures can be exactly reproduced. In contrast, non-permanent faults occur randomly and affect the system behavior for an unspecified period of time. The detection and localization of non-permanent faults is extremely difficult and they are often not uncovered during systematic testing. As shown in fig. 5, software faults are often classified into *Bohrbugs* and *Heisenbugs* [13].

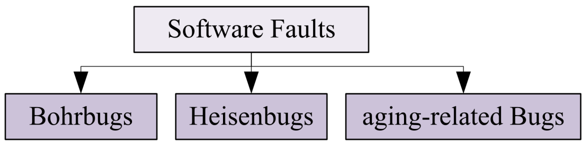


Figure 5. Fault Classification

Bohrbugs are permanent design faults whose resulting failures can be reproduced, thus are rather easily uncovered and mostly fixed in early testing phases of the software life cycle.

Heisenbugs are also essentially permanent faults, but their conditions of activation occur rarely or are not easily reproducible. The resulting transient failures may not recur after the software is restarted. *Heisenbugs* are therefore extremely difficult to identify through testing. Transient failures due to *Heisenbugs* are of special importance in multithreaded systems, which are usually considered non-deterministic. In a non-deterministic system, it is impossible to predict all the possible system states. It is obvious that a permanent software fault underlying complex activation conditions, in other words a very special system state, is extremely difficult to uncover. Race conditions are a typical example of *Heisenbugs* in multithreading environments. In mature software systems, failures caused by *Heisenbugs* are more likely than failures caused by *Bohrbugs*.

Another category of transient software failures is the manifestation of transient hardware faults. Modern microprocessors for example are less reliable and more susceptible to transient faults. This is largely due to faster and denser transistors on chip with lower threshold voltages and tighter noise margins [14]. These faults are not permanent faults but may result in incorrect program execution by inadvertently altering processor states, signal transfers or register values etc. The effect of these failures resembles the effects caused by *Heisenbugs*.

Another bug category is based on the well-known phenomenon that software systems running continuously for a long time tend to show a degraded performance and an increased failure-occurrence rate due to resource corruption over time.

3.2. Failure Model

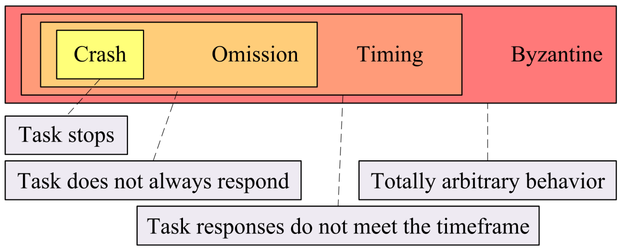
A failure model describes the type of failures that might happen and how the system behaves when the failure has occurred. While there are slight discrepancies in literature regarding their definitions [15], in this work the following failure models are used [16]:

Crash Failures happen if a failed process stops permanently at a certain time. This models a crash of a process that does not recover.

Omission Failures occur if a process continues its execution, but does not always respond to the inputs. The faulty task sometimes omits a response.

Timing Failures also model a scenario where the faulty task continues. In this case however it does not omit replies but responses are sent either too early or too late.

Byzantine Failures refer to a model where no assumptions about the behavior of a faulty process are made. A process can behave totally arbitrary and in the worst case do everything possible to compromise the system.



Crash
Omission
Timing
Byzantine
Task stops
Task does not always respond
Totally arbitrary behavior
Task responses do not meet the timeframe

Figure 6. Failure Model

Failures can occur in value- and/or time-domain, allowing a classification into valid replies (correct in timing and value) and invalid replies (invalid in timing or value). It is important to note that the first three failure models are based on time-domain failures, while all kind of failures in the value-domain are categorized as Byzantine Failures.

The presented models refer to failures on the level of processes. However, a failure of a process does not imply a failure on system level. It is a major goal in the design of self-healing systems to avoid failures on lower levels cause failures on higher levels.

4. System Analysis

4.1. System Model

An embedded system consists of both hardware and software components that interact with each other over certain interfaces. On the hardware level this is usually done using input/output lines or signals buses (Figure 7), on the software level the interaction between hardware- and software-components is done using operating system mechanisms, such as message queues, semaphores, events, etc.

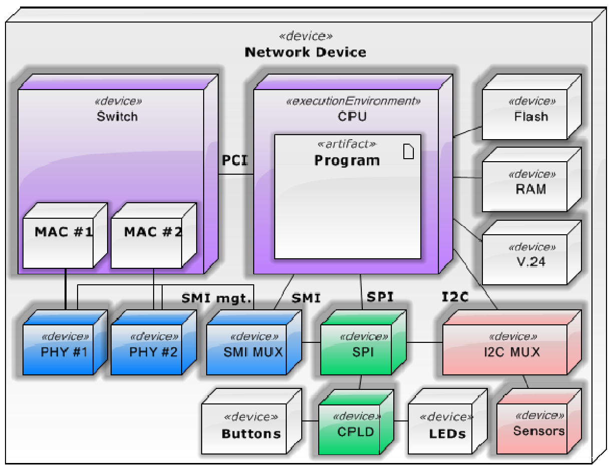


Figure 7. Hardware Architecture

We combine these basic models to derive the model of a concurrent system where processes communicate with each other and work on shared data objects. Inter-process communication mechanisms such as message queues and semaphores are basically also shared data objects (Figure 8). We will call this the “Component Interaction Architecture model” (CIAM) and base our system diagnosis analysis on this model.

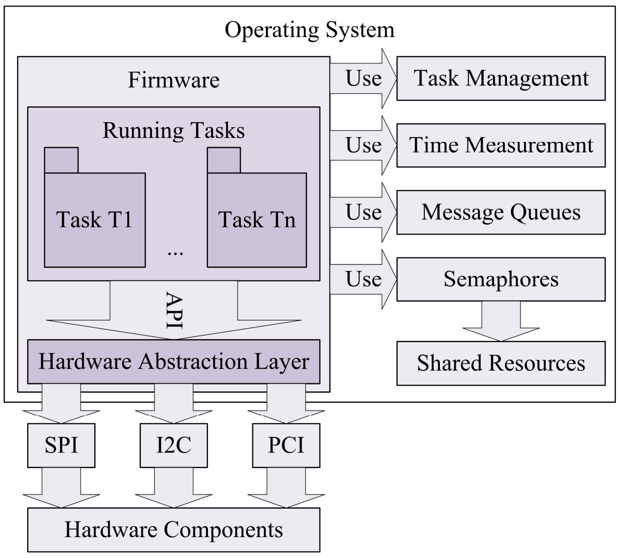


Figure 8. Component Interaction Architecture Model (CIAM)

Besides this interaction model, it is important to regard the behavior of the software during runtime. As a multithreaded system, the software is composed of a group of tasks, which are executed simultaneously. Resulting from the limitation of a single-core CPU, the amount of available CPU time is separated and assigned to the tasks by the operating system scheduler (Figure 9), which is necessary to support multitasking. With this method, the system simulates a concurrent execution of multiple tasks on a single processor.

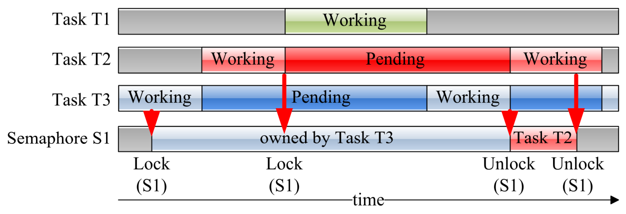


Figure 9. Dynamic Software Architecture

Depending on the task priority, the scheduler can also induce context switches and interrupt the currently running task in favor of another task with a higher priority waiting for systems resources. This method ensures that a system can always handle high priority requests.

4.2. Root Cause Failure Analysis

Based on historical data in a bug tracking database, a root cause analysis was conducted to identify the key elements of the system which are susceptible to the most expensive failures. This analysis revealed that the most critical parts of the embedded system are the mechanisms of the system that are responsible for communication between subsystems. Most of them are provided by the operating system: memory management, task synchronization through semaphores, inter-task communication through message queues etc. These are the elements of the system model (Figure 8) which are of special interest when developing a SSD system.

5. Fault Detection Methods

A self-healing software system is capable of continuous and automatic monitoring, diagnosis and remediation of software faults. Such a system's architecture is generally composed of two high-level elements: the software service whose integrity and availability is supervised and the components of the system that perform the monitoring, diagnosis and healing.

Because it is necessary for faults to become active to make their detection possible the first stage of runtime fault tolerance is always a mechanism founded on error or failure detection.

Fault detection mechanisms can be integrated into a system at two positions [17]: first, self-checking fault-detection mechanisms perform tests on the own module and review its internal state. Second, fault detection can be performed on other modules: self-protection mechanisms or acceptance tests are used to protect a module from faults originated from outside of the module. If not detected, these faults may induce a faulty state inside another module.

5.1. Error Detection in Time-Domain

Time-domain error detection algorithms are commonly used to monitor external modules, i.e. to perform acceptance tests.

An error detection algorithm in time-domain can be categorized by its ability to cover a set of failure models. It is important to note that an algorithm which is able to detect failures of one class also covers all underlying failure classes (Figure 6). That means that an algorithm which can reliably detect timing failures also detects crash and omission failures. This hierarchy is represented by the following statement, where "<" indicates that an algorithm which is able to detect failures of the class on the right also covers the failure class on the left side:

Crash Failure < Omission Failure < Timing Failure < Byzantine Failure

5.2. Error Detection in Value-Domain

Besides the error detection algorithms which can be applied in time-domain, errors can also be detected by performing checks on the data itself. The following types of error detection methods are commonly considered [18]:

Reasonableness Checks are used to verify if the data of a reply makes sense, e.g. by performing threshold checks or checking logical connections.

Structural Checks use known properties of module replies to detect errors. Some data types, such as lists, tables or trees can be verified for their correct logical organization. Structural test verify these data types for completeness, correct and valid referencing. A well-known example of structural tests is a file system check.

Coding Checks perform error detection using redundant information provided along with the data. Checksum codes, such as cyclic redundancy checks (CRC), are a type of coding checks. Coding checks can not only be applied to single data types, but can be used to check the integrity of a whole program.

Reversal Checks are a helpful error detection method if the backwards computation of an operation is much easier than the forward calculation. If backward calculation based on the output values of an operation results in the same input values given to the forward computation, the operation was successful.

Run-time Checks using hardware exceptions are a well-known and commonly used standard failure detection mechanism. Run-time checks provide application independent generic fault detection. A hardware exception is generated by the CPU during the execution of invalid instructions, such as division by zero, page faults or stack exception. Systems featuring a memory protection unit (MPU) also provide the capability to detect invalid memory accesses when separate address spaces are used. Memory protection can detect and isolate faulty programs which try to access invalid memory areas, e.g. by dereferencing dangling pointers.

Besides the methods proposed by [18], the following methods are also commonly considered as a mean to detect failures:

Static Redundancy using software diversity (n-version programming) is an error detection mechanism in which the monitored function is implemented several (n) times using different approaches, e.g. different programming languages, algorithms, development teams, etc. During runtime, these implementations are executed simultaneously, followed by a comparison [19].

Control Flow Monitoring is an error detection method which monitors the program execution and ensures that only allowed instruction sequences are executed and only allowed branches are taken. The control flow monitoring method requires information about the allowed control flow paths, which can be defined manually or by collecting data during a test run. Control flow monitoring can also make a program tamper-proof. By verifying each basic code block using signatures, control flow monitoring provides powerful means to stem software cracking [20], thus increasing system security.

6. Error Detection Mechanisms

By today's standards, most software projects already use a variety of measurements to achieve dependability:

Fault-Prevention and Fault-Removal strategies are commonly applied, e.g. by using coding guidelines or perform system tests. Fault-tolerance and fault-forecasting strategies, however, are so far not widely used.

Since a root cause failure analysis on historical data has revealed that the most critical parts of the embedded software are the mechanisms responsible for sub-system communication, our SSD solution will enhance the critical operating system mechanisms by applying the following error detection mechanisms:

6.1. Task Monitoring

The task management mechanisms provided by the operating system allows designing the software system as a number of concurrently running tasks. As shown in the failure model presented previously, the behavior of a faulty task is unpredictable, but different fault detection algorithms can be used to detect a subset of these failures.

Our solution tackles the problem of detecting failed tasks by using a time-domain error detection algorithm, which is able to detect timing failures therefore also covering omission and crash failures. A task is requested to send regular heartbeat messages to indicate that it is still running. If a heartbeat message is invalid or missing, the monitored task is assumed to have failed.

6.2. Communication Mechanism Monitoring

Synchronization and communication mechanisms, e.g. semaphores, message queues, events or timers are prone to error propagation and thus a critical part of a system. Consider a message queue which overflows because a faulty task does not receive messages, making communication between other tasks impossible. Another example are accidentally locked semaphores, e.g. due to a crashed task which failed to release them.

These problems demonstrate that run-time supervision of these mechanisms is necessary. Our approach supervises these mechanisms using two methods:

Online Error Detection is used to detect locked access control (mutual exclusion) semaphores. They are monitored online by adding a time-domain error detection algorithm. Similar to the method used for task monitoring, a task which owns a mutual exclusion semaphore used for access control is requested to send regular heartbeat messages to indicate that it still accesses the shared resource. If a heartbeat is omitted, appropriate recovery measures can be deployed.

Integrity Checks mitigate the risk of possibly occurring data corruption, e.g. when shared data objects are used in multithreaded systems. If such data is accessed by two or more threads at the same time without access control, data corruption can likely occur. Integrity checks can be used to handle such situations, e.g. by extending the object with some sort of redundancy (checksums) or by using robust data types. Our solution validates the data using a magic number check to detect invalid objects.

6.3. Application-Specific Self Tests

Error detection using a generic algorithm is often not sufficient. To perform detailed error detection on software modules, knowledge about the internal structure is required to perform meaningful error detection. In fact, most of the value-domain fault detection algorithms mentioned can only be deployed as case-by-case tests. Thus, the third error detection mechanism of our SSD solution is based on ad-hoc self-tests.

7. Failure Recovery

If a failure in a system occurs and is diagnosed or can be predicted to occur soon, it is necessary to initiate an action to correct or prevent the failure from affecting the system to a degree that would make normal system usage impossible.

A widely used method to recover from transient failures in complex software systems is to reboot the whole system. While this may be a method that can easily be applied, the disadvantage of this approach is that the downtime caused by a complete system reboot directly contradicts the requirement of high system availability for industrial automation systems. For this reason, other approaches are to be found, taking into account the availability requirements of industrial systems.

One such alternative approach is to selectively restart only the sub-system, where the failure has occurred. In order to allow a selective restart of affected sub-systems, several conditions have to be observed. This is non-trivial and not the focus of this paper, thus we will not go into further detail on possible approaches for system recovery on sub-system level. Existing approaches to solutions for this problem are already profoundly covered in other publications. The focus of these approaches varies from the recovery of faults at the device driver level [21] to the recovery of failures in multiprocessor systems or distributed computer grids [22], [23] or [24].

As a final remark on failure recovery it has to be noted that in some fields in the industry partial system recovery is not an option. This is especially true for safety critical systems. Due to legal and normative guidelines, in such environments the safety of a system has to be guaranteed at all times. If a failure occurs in the safety critical system, it has to assume a safe state and has to remain in this state until the failure has been removed and safe operation can be guaranteed again. While it might be possible to design systems that still can recover on a sub-system level, the effort involved in certifying such systems usually is too high to make this feasible.

8. SSD Framework

The integration of the error detection and recovery mechanisms into an existing software system resulted in the development of a library to address self-diagnosis and fault-tolerance requirements. The essence of the library is a rule-based expert system [25], consisting of a behavioral logic, the inference engine, and a rule base (see Figure 10).

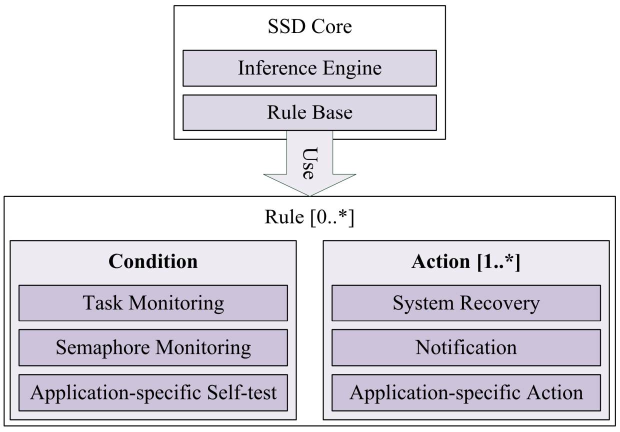


Figure 10. SSD Framework Architecture

To extend a software service with self-diagnosis and self-healing abilities, the library's user creates rules to specify how errors are detected and how they are treated. These rules are then stored inside of the rule base of the expert system, from where they are monitored by the inference engine. Each rule consists of a condition and one or more actions.

The condition poses the error detection mechanism of the rule. Based on the strategies described in section 6, a condition can be configured to monitor a task for responsiveness, to supervise an access control semaphore or to perform a module self-test. Each rule supports only one condition, it is therefore impossible to use several error detection mechanisms within the same rule.

To complete a rule for the SSD framework, the error detection mechanism is combined with a recovery strategy, which specifies the behavior of the SSD functionality in case the condition identifies a faulty state. The implementation of the recovery strategy is done by dividing it into a set of single operations which, within the context of the SSD framework, are called actions. These actions are then added to the rule. The number of actions per rule is not limited, thus it is possible to implement more or less complex recovery strategies. By correctly defining the set of actions, it is possible to restart different sub-systems even if they are dependent on each other by stopping and restarting them in the proper sequence.

The library supports three major types of actions: system recovery, notification and application-specific actions. System recovery actions provide methods to recover the device in a generic fashion using operating system function. These include selective restarting of tasks, restoring communication mechanisms or rebooting the whole device. The second type is used for debug and "Design for Testability" purposes: notification actions are used to provide failure information. This action type can be used to validate the hardware after manufacturing or to store error information of rarely occurring *Heisenbugs* after deployment. The third recovery strategy is necessary if generic recovery approaches are impractical: similar to self-tests, the library also supports the definition of application-specific actions. In this case, the library's user himself implements the necessary recovery measures.

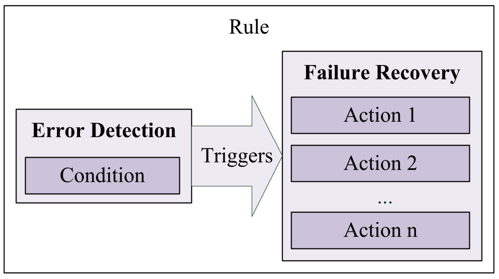


Figure 11. SSD Framework Rule Structure

During runtime, the rule-based expert system is executed as an additional system task. The rule base is checked periodically by the inference engine. If a faulty state is detected, the rule condition triggers the corresponding actions, which are then executed successively (Figure 11).

However, it cannot be guaranteed that a recovery strategy will be successful. The same failure may reoccur or a recovery mechanism proves to be ineffective. To address such issues, the behavior of the SSD framework can be customized depending on the attempt to recover from the same failure.

The usage of the SSD framework's functionality should be made compliant in the development process, forcing the software engineers to think about possible faults and failures from the very beginning in the design and implementation process, thus increasing the likelihood of preventing them in the first place. All created SSD rules are applicable not only throughout the development phases, but in the overall product lifecycle. In this way, a "Design for Testability" approach with a very high leverage regarding efficiency can be achieved [2].

9. Summary and Conclusion

In this work, a generic model and methodological guidelines were developed for a software based self-diagnosis system and associated recovery strategies on embedded systems. This can be used as a basis for development of a related embedded diagnostics and recovery module.

Preliminary research in issue databases of complex embedded network products has shown that almost all expensive to debug transient errors had their root cause in the erroneous behavior of internal system communications mechanisms.

Many faults are detectable indirectly, in form of performance disorders that manifests as anomalies in monitored data. Anomaly detection is therefore a primary means for fault detection.

Diagnosis and prognosis are basic tools for anomaly detection, but they can simultaneously also improve system security. Thus the anomaly detection can also be used for intrusion detection, improving system security.

Not to be underestimated is the positive influence on the engineering design process that is created by the application of the SSD framework: The engineers are forced to think in quality terms ("What can go wrong?") throughout the whole design cycle. In this way, many faults will not even be coded into the software in the first place.

References

[1] C. Rojas, P. Morell, "Guidelines for Industrial Ethernet infrastructure implementation: A control engineer's guide" *Cement Industry Technical Conference, 2010 IEEE-IAS/PCA 52nd* , vol., no., pp.1,18, March 28 2010-April 1 2010

[2] M. Rentschler, "Faulty Device Replacement for Industrial Networks", INDIN 2012, Beijing, China, 2012

[3] M. Rentschler, "Design for Testability – The Holistic Future of Testing?" *in Testing Experience 12/2011, Nr. 16, available at www.testingexperience.com*

[4] C. Wehner, *Tornado and VxWorks*, Book on Demand, 2006. See also http://en.wikipedia.org/wiki/VxWorks

[5] J. C. Laprie, *Dependability: Basic Concepts & Terminology,* Springer-Verlag, New York, 1992.

[6] A. Avižienis, J. C. Laprie, B. Randell, "Dependability and its Threats: A Taxonomy", *Building the Information Society*, Topic 3, pp. 91-120, 2004.

[7] A. Avižienis, J.C. Laprie, B. Randell, C. E. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transactions on Dependable and Secure Computing*, 2004.

[8] R. Natella, D. Cotroneo, "Emulation of Transient Software Faults for Dependability Assessment: A Case Study", *Proceedings of the 2010 European Dependable Computing Conference*, pp. 23-32, 2010.

[9] E. W. Dijkstra, "Notes on Structural Programming", *T.H.-Report 70-WSK-03 second edition*, 1970.

[10] C. Buckl, A. Knoll, G. Schrott, "Model-based Development of Fault-Tolerant Embedded Software", *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 103-110, 2006.

[11] C. Walter, M. M. Hugue, N. Suri, "Continual On-Line Diagnosis of Hybrid Faults," *IEEE Transactions on Software Engineering*, Vol.23, No.11, pp. 684-721, 1997.

[12] C. Walter, P. Lincoln, and N. Suri, "Formally Verified On-Line Diagnosis," *IEEE Trans. Software Eng*., Vol.23, No.11, pp. 684-721, 1997.

[13] J. Gray, "Why Do Computers Stop and What Can Be Done About It?", *Technical Report 85.7.*, Tandem Computers, pp. 17-19, 1985.

[14] P. Shivakumar, M. D. Kistler, S. W. Keckler, D. C. Burger, L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic", *Proc. of the 2002 International Conference on Dependable Systems and Networks*, pp. 389 – 398, 2002.

[15] M. Barborak, M. Malek, A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing", University of Texas, Austin, Texas, USA, 1991.

[16] F. Christian, H. Aghili, R. Strong, D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", *Fifteenth International Symposium on Fault-Tolerant Computing*, pp. 200-206, 1985.

[17] W. Torres-Pomales, "Software Fault Tolerance: A Tutorial", Langley Research Center, Hampton, Virginia, USA, 2000.

[18] T. Anderson, P. A. Lee, *Fault Tolerance: Principles and Practice*, Prentice/Hall International, 1981.

[19] L. Chen, A. Avižienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation", *Proceedings of FTCS-8*, pp 3 - 9, 1978.

[20] M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti, "Control Flow Integrity", *ACM Transactions on Information and System Security*, Vol.3 Issue 1, No.4, 2009.

[21] M. Swift, M. Annamalai, B. Bershad, H. Levy, "Recovering Device Drivers", *ACM Transactions on Computer Systems*, Vol.24 Issue 4, pp. 333-360, 2006.

[22] J. Hursey, J. Squyres, T. Mattox, A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI", *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, 2007.

[23] Zizhong Cheng, J. J. Dongarra, "Algorithm-Based Checkpoint-Free Fault Tolerance for Parallel Matrix Computations on Volatile Resources.", *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[24] G. Candea, Shinichi Kawamoto, Yuichi Fujiki, G. Friedman, A. Fox, "Microreboot – A Technique for Cheap Recovery", *USENIX Association OSDI'04: 6th Symposium on Operating System Design and Implementation, San Francisco, USA, December 2004.*

[25] A. Abraham, "Rule-based Expert Systems", *Handbook of Measuring System Design",* edited by Peter H. Sydenham and Richard Thorn, John Wiley & Sons, Ltd., ISBN 0-470-02143-8, 2005.