Proceedings

IEEE INDIN 2008

6th IEEE International Conference on Industrial Informatics

July 13-16, 2008
Daejeon Convention Center
Daejeon, Korea

Table of Contents

IEEE Catalog Number:
CFP08INI-CDR
ISBN:978-1-4244-2171-8
ISSN:1935-4576
¨Ï2008 IEEE

Adobe Reader 7.0
is recommended for best viewing

Safety Supervision Layer

Georg Hartner, Andreas Gerstinger
Institute of Computer Technology, Vienna University of Technology, Austria
georg.hartner@gmail.com, gerstinger@ict.tuwien.ac.at

Abstract - **This work covers a generic approach to fault detection for operating systems in fail-safe environments. A safety supervision layer between the application layer and the operating system interface is discussed. It is an attempt to detect operating system and hardware faults in an end-to-end way. Standard POSIX system calls are wrapped by procedures that provide fault detection features. Furthermore, potentials of an additional watchdog module on top of the operating system interface are analyzed. Applications that use the Safety Supervision Layer are notified of detected faults and deal with them by providing specific handlers to bring the fail-safe system to its safe state. The goal of the presented layer is to encapsulate the operating system and hardware layers a safety-critical application resides on, in order to detect faults produced by those and bring the system to a safe state. Advantages of such an attempt are portability, lower time-to-market, higher cost efficiency in building fail-safe systems and – most important – reduced error detection latency compared to usual periodic supervision approaches.**

I. Introduction

Fail-safe systems know a safe state, i.e. a static set of output values, in which the system cannot cause any harm [1]. If the system or some of its sub-components yield calculation results differing from their specification, this is seen as an error. The correct operation cannot be guaranteed any more, hence a safety reaction has to be initiated – the system has to be brought to its safe state.

A. Example

As this work was done considering fail-safe railway applications, the project environment yields a good and simple example (Fig. 1): Two trains go in the opposite direction, between them there is a track switch that is supposed to lead the first train to a free track and not to the one the other train is going on. The railway control application makes decisions based on wrong calculations by operating system calls and sends the two trains to the same track – a hazard which is likely to lead to a crash. The presented layer *does not try to avoid* such wrong calculations, but it tries to *detect such errors* before they can cause a hazard and bring the system to a safe state. In the example it would just stop the two trains. Hence this approach aims strictly to safety and not to reliability and availability considerations.

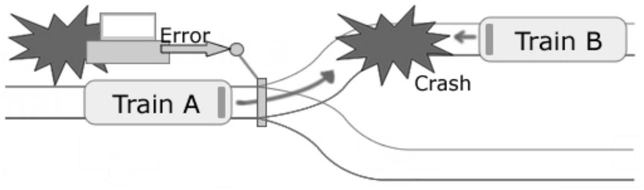

Fig. 1.: *Visualization of the basic example.* An error produced by the operating system leads to wrong positioning of the train switch, which leads to a crash of the trains.

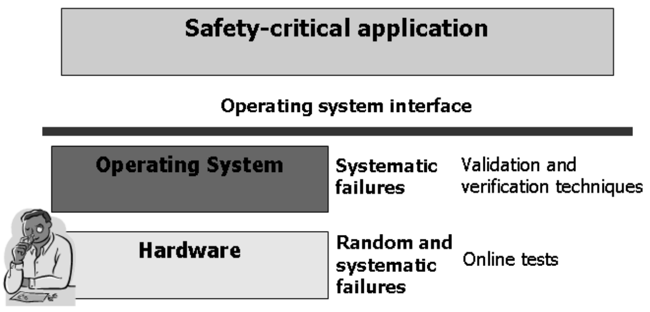
Safety-critical application
Operating system interface
Operating System
Systematic failures
Validation and verification techniques
Hardware
Random and systematic failures
Online tests

Fig. 2.: *Typical setup for safety-critical applications considered in this work.* The safety-critical application accesses operating system functionality through the operating system interface. Operating system reliability is increased by using validation and verification techniques. Hardware functionality is checked by periodic online tests.

B. Usual Application Setup

In a typical setup for the systems considered in this work, a safety-critical control application resides on top of the operating system interface (Fig. 2). Thus hardware and operating system have to be considered as safety-critical components. Hence calculations and decisions made by the application may be based on erroneous results yielded by those lower-level layers, they have to be developed according to the same standards as the application itself.

C. Problems according Operating System Assessment

Currently, operating systems in use are developed according to safety standards. Quality assurance techniques like validation, rigorous testing and code reviews are applied to the system. Every new software version has to be verified separately line by line, tests have to be adapted or re-written, which is a very tedious and expensive task. This is especially a hurdle for the application of open source systems that underlie unsteady version updates. There exist studies assessing the applicability of Linux in safety-related environments, e.g. [2], but they can only make spongy assertions in a general matter, sometimes only justified by proven-in-use.

An example for such a statement is given in [2]: "*On the basis of evidence from widespread use, some numeric reliability data, observed reliability growth, the existence of test projects and the limited analysis carried out by this study, it is concluded that "vanilla" Linux would be broadly acceptable for use in safety related applications of SIL 1 and SIL 2 integrity*". They state that it *might be feasible* to certify Linux to SIL 3 in an IEC 61508/61511 context [2].

The main reason for why no clear statements can be done in this general way is, that this current approach to operating system assessment is a reliability one. Their main goal is to guarantee that the operating system works properly in all situations. As the main goal is to ensure safety and the considered environments are fail-safe, the claim can be reduced to detect errors produced by layers below the operating system interface. When such errors are detected, the system shall be brought to the safe state. This claim is easier to satisfy, as calculation results can be clearly checked against the according system call specification.

D. *Problems according Hardware Tests*

Hardware components are usually checked during operation by online tests. When erroneous functionality is detected, the system is notified and supposed to turn into the safe state. These tests are often hardware specific and have to be re-written for every new hardware version. Generally, two types of problems are identified for this approach.

The problem with every periodic assessment approach is, that there exists a time span between two points in time a test run is completed. If a fault occurs shortly after such a point in time it is not recognized until the next test run. This leads to a *possibly high fault detection latency* and harms the system's determinism.

The second problem is that those tests just check the hardware *components*, but there is no guarantee for the *interfaces* between them. As applications use provided functions, an advantageous approach would be to evaluate a yielded result directly against what can be expected because of the specification in an end-to-end way.

E. *Related Work*

Reference [3] defines common terms and techniques that may be used to construct dependable computing systems. Reference [4] provides an overview and justifications of relevant software fault tolerance techniques.

Reference [5] gives a number of definitions of Off-The-Shelf (OTS) components and explains potentials and possible pitfalls for systems making use of them. Apart from that it shows possibilities of how to guarantee Safety Integrity Levels in those systems. Reference [6] deals with the reuse of Commercial-Off-The-Shelf (COTS) components and Software of Unknown Pedigree (SOUP). The fact that safety standards have strict requirements to how software has to be developed and tested, and most do not explicitly allow the use of COTS components or SOUP is stated and ways to deal with these circumstances are stated. Reference [7] gives a schematic overview of what has to be considered, when selecting components for safety-related systems from market monitoring to COTS Acquisition contract. Reference [8] examines ways to analyze the criticality of COTS software components using commonly known technologies like System Hazard and Fault Tree Analysis. The watchdog module presented in chapter IV deals with ways to check for failures that cannot or can hardly be detected directly when using the respective functionality. Many of them deal with hardware and low-level operating system faults. References [9] and [10] deal with testing, error detection and fault analyses for those.

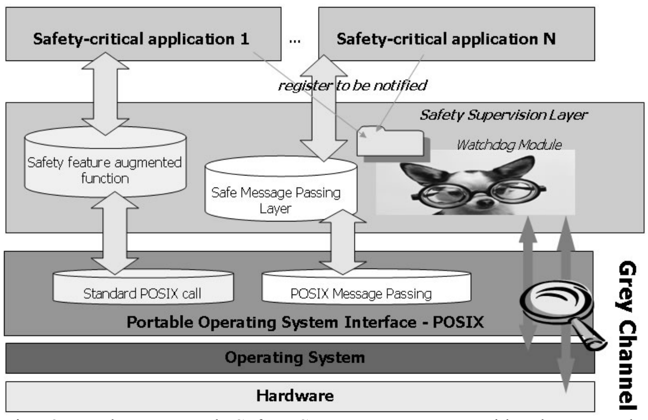
Safety-critical application 1
...
Safety-critical application N
register to be notified
Safety Supervision Layer
Watchdog Module
Safety feature augmented function
Safe Message Passing Layer
Standard POSIX call
POSIX Message Passing
Portable Operating System Interface - POSIX
Operating System
Hardware
Grey Channel

Fig. 3.: The proposed *Safety Supervision Layer* resides between the application and the operating system interface. It consists of two parts: a safe system call library and a watchdog module. This setup allows the application developer to see the layers below as a grey channel.

Reference [2] states requirements to safe operating systems assessing Linux. It states techniques and issues heavily using proven-in-use that have to be used when deciding whether a given operating system may be used in safety critical environments. They also show how the results of such assessments can be mapped to safety integrity levels. More generic sources are provided by the CENELEC standards [11], [12] and [13], that exactly prescribe required features in the railway problem domain.

II. SAFETY SUPERVISION LAYER – OVERVIEW

Fig. 3 shows the conceptual design of the proposed layer. The safety-critical application accesses functionality provided by that layer and does not use operating system or hardware functionality directly.

A. *Layer Architecture*

Basically the layer consists of two parts: the safe system call library, which is the main part, and a watchdog module that performs functional tests periodically. The safe system call library is an application programming interface that wraps standard system calls. The main focus lies on this structure. Detection of faults that can be covered by it should be implemented in this library, because it provides a direct end-to-end way to fault detection. This yields important benefits compared to the periodic tests of the watchdog module as explained below.

The Safety Supervision Layer provides notification features to inform interested applications about potential errors. As shutdown procedures may be complex and the layer itself does not know anything about the system structure above, the applications have to provide error handlers themselves to bring the system to a safe state. E.g. in the case sketched by Figure 1, a single switch control node does not know how many trains are on the whole track that have to be stopped, when an error occurs, so this specific problem has to be solved by the higher level control system.

B. The Grey Channel

The application shall be able to consider the lower layers as a *grey channel* from a safety point of view. This means that the application does not need to know anything about the implementation of those lower levels and that shall be able to fully rely on them. To clarify this definition in the presented *safety-centric* approach: it can not be sure that they are always working properly, but that occurring errors are detected and a notification is produced. Further explanation can be found in the safe message passing example in chapter III that deals with *safe communication over unsafe infrastructure*.

C. Portability and POSIX Compatibility

One main focus of the layer is to provide the application developer the possibility to make the lower level layers *interchangeable* from a *safety point of view*. To support this goal, the layer is based on POSIX (Portable Operating System Interface, [14]), which is provided by most operating systems. To guarantee high portability, the safe system call library wraps POSIX functions and tries to imitate their usage and behavior as far as possible.

The standard watchdog module's conception puts its focus on the usage of POSIX functions, but the flexible conception allows the embedment of supervision for proprietary functionality.

III. Safe System Call Library

For the design of the of the safe system call library, the specifications of system call functions have to be analyzed augmented with suitable fault detection capabilities.

Applications use the new functionality exactly the same way as they use the common POSIX functions. By using preloading, this can be made transparent to the application. The augmented functions implement fault detection algorithms to get detect faults in an end-to-end way directly at the use of the respective call.

A. Basic Procedure

In this context libraries may deal with operating system functionality like message passing, process synchronization, timing functionality and a lot of other functions that may be utilized using POSIX. Applications using these libraries are notified immediately after an error is detected, as all results produced are checked. As stated above we need to prove that a given function covers all possible faults, so every library module has to be proven using common technologies like FTA and FMEA.

B. Restrictions

In most cases, applicable standards define prerequisites to a given environment. For example, CENELEC 50159-1 [15], that was used as a basis for the safe message passing library described in section IV, prescribes methods for dealing with closed communication systems. Closed communication systems means, that the communication media is known and specified, that only authenticated access is allowed and there exists a known maximum number of communication participants.

As many usual functions provided by the operating system interface provide more flexibility and do not obey such clearly defined prerequisites, specific restrictions have to be specified. E.g. in the safe message passing library it has been defined, that the library does only support 1:1 communication due to the authentication and sequence check requirements of [15]. If the application tries to violate the given restrictions in the initialization phase, this must be recognized as an error, as states out of that clear specification have to be avoided.

C. Layer Embedment

Applications simply include and use the layer libraries like the usual ones. The usage should be as similar as possible to the common functionality. If restrictions are violated, this has to be recognized by the respective library functions. Using such well-defined libraries, the application can now be sure that the result yielded by those procedures is either correct or it is notified of an error. Therefore the application has to provide error handlers that consume those signals and provide methods for the transition to the safe state. Fault analyses the detection methods are based on have to be examined in the documentation.

D. End-to-End Checks vs. Periodic Tests

The safe system call library approach provides important advantages compared to the periodic test approach described in the introduction and in chapter V. Periodic checks can only guarantee or deny correctness for the whole system or components at *specified points in time*, when a *test run* is complete. Fig. 4 describes a case in which this may not be sufficient. A fault occurs shortly after a checkpoint. This may happen, e.g. when some hardware component begins to fail. In the periodic check case, the fault would not be detected until the next test run. The time span between the fault occurrence and the next check point is critical, as correct system operation cannot be guaranteed until then. The safe system call library does not know such check points, but *checks the results of calculations against plausibility criteria*. Thus this *fault detection latency is minimized* by this approach.

Another important fault class that may not be detected by periodic checks are transient faults, e.g. short-time hardware dysfunctions. If they happen between two checkpoints, they would simply not be detected, even at the next test run.

A problem with the end-to-end approach may be the

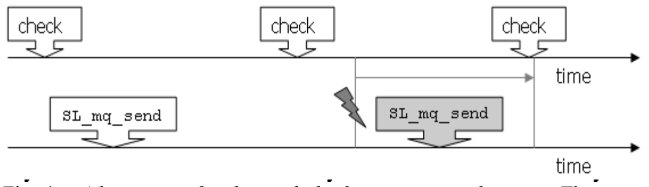

Fig. 4.: *Advantages of end-to-end checks versus periodic tests*. The upper timeline visualizes a main problem. A fault occurs shortly after a checkpoint. Possible faults are not detected until the next test run. The lower timeline shows that the end-to-end approach solves this problem, as it eliminates this latency.

structural and calculation overhead it causes. Calculation results are always checked against plausibility criteria. Thus there may be negative effects a system's performance. Periodic check complexity may clearly be calculated, their resource consummation may be constant and they can be suspended in a peak load case. This is not reasonable in the end-to-end approach.

IV. Example - Safe Message Passing

For a detailed analysis of the presented approach, a *safe message passing library* has been designed and implemented. This library wraps the usual POSIX message passing functionality and provides e.g. sequencing, transmission error detection and feedback infrastructure.

The POSIX standard ([14]) defines *inter-process communication* structures based on message queues. In literature those are known as POSIX message queues and are based on System V message queues, in which the processes can communicate with one another by means of messages. Each message is sent through an IPC message queue, where it remains until another process reads it. When the other process reads – or consumes - it, the message is deleted. A message is composed of a fixed-size header and a variable length body. The characteristic that the message is destroyed by the kernel leads to the fact that only one process can receive a given message.

A. Fault Analysis

To choose proper techniques for detecting errors related to message passing, possible faults have to be analyzed in detail. This was done by *fault trees* and *failure modes and effects analyses*. Fig. 5 shows an example of a fault tree used to examine a fault case. Additionally, failure modes and effects analyses have been examined to analyze the criticality of the considered faults. The results of these analyses were compared to the considered standard.

B. Justification – CENELEC 50159-1

Reference [15] defines methods for safe communication over unsafe infrastructure, while error detection is handled by safety procedures. Inter-process message passing was seen as a closed communication system, with the operating system considered as the unsafe infrastructure. According to [15], the following prerequisites have to be fulfilled in order to consider a system as a closed communication system:

- Only authenticated access is allowed.
- There exist a known maximal number of communication participants.
- The transmission media is known and properly specified.

POSIX message passing does provide more flexible functions and does not fulfill the given prerequisites. This is why an important restriction had to be defined. The safe message passing layer does only define one-to-one communication. So the known maximal number of communication participants is now 2, only communication from one process to another one is supported. If an application tries to violate this restriction, this is considered as an error.

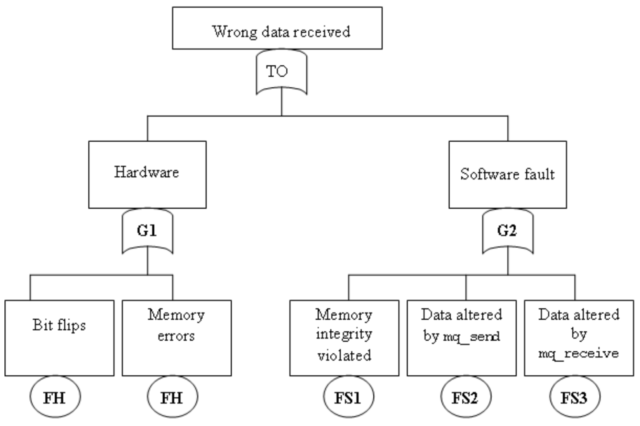

Fig. 5.: *Example of a fault tree for the case* "*Wrong data received*". This error is deduced to its potential causal factors.

C. Resulting Features

The measures that resulted from the analyses are in accordance with the measures required by the standard [15]. They are:

1) *Feedback infrastructure:* To be able to determine whether a message has not been consumed by the sender or to decide whether a message has just not been consumed within defined time constraints, a communication channel back to the sender is needed. In that feedback messages receiver and sender identification, sequence numbers and checksums are included again.
2) *Sender and Receiver identification:* An authentication system is provided by the `mq_open` function. When the sender opens the message queue, it generates an identifier for itself. The same is done by the receiver, who sends a "Hello" message to the sender, just after the initialization.
3) *Sequencing:* To determine whether the messages arrive in the right order, the Safe Message Passing functions sequence each message. The first message gets the sequence number INT_MAX-1 to ensure that value overflow works properly. The sender increases its sequence counter after each send, the receiver checks the received message sequence number and increases its own after consummation.
4) *Checksums:* A checksum is computed over the whole message payload containing the sender and receiver identifiers, the sequence numbers, timestamps and data type verifiers. This provides to a counter measure to common mode failures.
5) *Periodic supervision of the communication channel:* Even if there's no planned communication test messages are sent through the channel to ensure that it is working. This was not a feature required by the standard but demanded by the project partner.

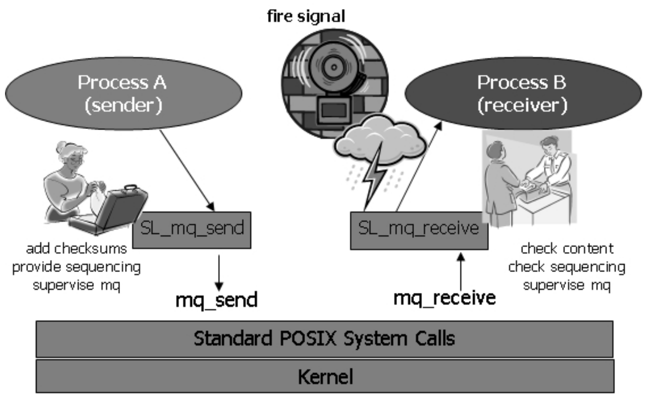

Fig. 6.: *Basic error detection strategy of the safe message passing library*. The sender side generates the data for the plausibility tests performed by the receiver. Feedback is checked by timed functions.

D. *Communication Process*

Fig. 6 shows the adapted communication process. The sender of the message primarily augments the message payload with redundant data needed for the plausibility checks. The receiver side checks the message content against those plausibility criteria and raises a signal, if an error is detected. In this specific implementation the sender process additionally listens for a feedback message for a given tolerance time span and supervises the communication channel.

All the communication on the raw message queues is done blocking. The safe message passing layer prototype wraps the following three functions:

1) mq_open: This function is the most complex, as it sets up the communication infrastructure. For each side it establishes one message queue for sending and one for receiving feedback. Every process has its own role: it is either the creator or the consumer of the message queue. When both of them open the queue, they perform a two-way hand-shake protocol to identify themselves. If a third process now tries to open the named queue, this would violate the 1:1 restriction, hence an alarm is raised. To detect this, the two processes identify themselves in this initialization scenario by their process ids. Those identifiers are sent with the message payload at every send and feedback procedure.
2) mq_send: The send function takes the message payload, sets the message type and puts a "valid through"-timestamp onto the message, as well as a sequence number. Then this is content is serialized, a checksum is computed over the whole message and it is sent through the raw queue. After sending, the process waits for feedback.
3) mq_receive: This procedure listens for incoming messages on the queue. Then it de-serializes the received item and performs checks against expected sender id, checksum, sequence numbers and timestamps. If those plausibility checks fail, an alarm is raised.

The requirement of a periodic supervision of the communication channel required a message dispatcher function that handles the reception of messages and automatically sends feedback. For a detailed description of the chosen implementation and further information about the communication infrastructure refer to [16].

V. The Watchdog

As the direct end-to-end approach described in chapters III and IV provides a number of advantages (see chapter III – D, IV – B), it will not be able to cover all possible faults this way. Reasons for this may be the higher resource consumption or scenarios that require checks "from the outside". An example for the latter is that it is hardly possible to detect incorrect scheduling behavior from the inside of a process. A process won't be able to detect, whether it is starving. To cope with these circumstances a higher-level watchdog structure has to be included in the Safety Supervision Layer.

A. *Watchdog Architecture*

The watchdog is intended to be a user-level daemon that runs specified test modules periodically. Those modules can be loaded and configured flexibly via structured configuration files[1]. Options for the modules should be a delay for the first test run, period, priority and, optionally, fault class-specific signal handlers. Furthermore extensive logging and reporting functionality should be provided, as these can be used for offline infrastructural tests and analyses during operation. The watchdog was not implemented during the thesis.

B. *Implementation Proposals*

Currently there exist a great number of available offline operating system test checks. For example they cover memory management, scheduling, device drivers or process memory integrity. The Linux Test Project[2] is a framework that features modular tests. Those tests can be easily adapted and included by the watchdog.

C. *Analysis*

The difference to the usual periodic tests is that they reside on top of the operating system interface as well. This covers a broader range of faults than lower-level tests that may influence results of functions used by the safety-critical application. Component tests just cover faults from the inside, but they do not check the interfaces and side effects from other components. Another advantage is that those tests get portable. Usual periodic hardware tests are hardware specific and complex. They have to be re-written and re-verified for every new hardware used. Although there are advantages of higher-level periodic tests compared to the lower-level ones, efforts should be made to cover as many faults as possible with the safe system call library approach because of the reasons examined in chapter III.

[1] Modules should be loadable in a similar way as it is featured by the Apache Web Server. Further information: http://httpd.apache.org/docs/2.0/

[2] http://ltp.sourceforge.net

VI. Conclusion

The elaborated structures are suitable to implement fault-detection in safety-critical systems. Generally, they embody an approach to detect failures directly at the service interface, as the input is used to directly generate plausibility values for the generated output. From a safety point of view, this is at least as good as the current approach of validation and hardware tests, with the added benefit of higher hardware and operating system independence. The goal of the commonly used online tests is to detect faults before they are activated, which is a search for the needle in the haystack. The safe system call library checks for faults when the functions are used, which leads to the fact that faults are detected before they can cause a failure and ensures temporally deterministic fault detection with minimized latencies.

A. *Advantages of Higher-Level Supervision Structures*

The proposed structure provides a generic basis for operating system and hardware verification using a consistent end-to-end approach. The safe system call library as well as the periodic checks on top of the operating interface provide end-to-end error detection. Only checking the function's components' functionalities just ensures that the components work properly, but not if the whole system runs correctly. Apart from a supervision layer, the proposed structure can facilitate portability. In a mature version the layer can be a full substitute for tests that are currently written for specific operating systems and hardware.

B. *Direct End-to-End Error Detection*

The basis for the plausibility checks are the specifications of the functions the layer wraps. Generally, results are checked before they are returned, hence it can be guaranteed that it is correct. In contrary to a periodic approach, the system does not suffer unpredictable error detection latency. A disadvantage is, that such an approach leads to a structural and temporal overhead.

C. *Higher-Level Periodic Tests*

Higher-level tests check a given result of a performed operation against plausibility criteria. They test the whole procedure from the function call to the code and the hardware used by the call. This means that concerns about interfaces between the lower level layers and their tests may be neglected, if the layer itself is written properly. Only checking the function's components' functionalities just ensures that the components work properly, but not if the whole system runs correctly.

D. *Outlook*

The proposed layer is a starting point for the decomposition of safety-critical applications and the infrastructure they run on. The possibility to change the underlying hardware and operating systems without harming safety are obvious portability advantages.

To provide a reliable structure, a complete fault analysis for operating systems should be done. Other functionality apart from message passing should be considered and analyzed using standards and fault analysis techniques similar to the ones described in chapter IV. The goal is to imitate the original functions as far as possible, to provide total transparency to the application implementers.

Acknowledgment

This paper resulted from the diploma thesis "Safety Supervision Layer – Fault Detection for Operating Systems in Fail-Safe Environments" [16]. It was elaborated at the Institute of Computer Technology, Vienna University of Technology, under the guidance of Andreas Gerstinger.

References

[1] Neil Storey. Safety-Critical Computer Systems. Addison Wesley, 1996, p. 22.

[2] Ron Pierce. *Preliminary assessment of Linux for safety related systems*, ISBN: 0717625383, International Limited for UK Health and Safety Executive, August 14th, 2002.

[3] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr. *Basic concepts and taxonomy of dependable and secure computing*, IEEE Transactions on Dependable and Secure Computing. Volume 1, Issue 1, Jan 2004.

[4] Laura Pullum. *Software fault tolerance techniques and implementation*, Artech House computing library, ISBN 1-58053-137-7, 2001.

[5] Andreas Aas. Essay on *Use of COTS Software in Safety-Critical Systems*, 2007, unpublished[3].

[6] Torbjørn Skramstad. *Assessment of Safety Critical Systems with COTS Software and Software of uncertain pedigree*, ERCIM Conference on Dependable Software intensive embedded Systems; Porto 30.08.2005 - 03.09.2005. Proceedings from ERCIM Workshop on Dependable Systems, 2005.

[7] Fan Ye, Tim Kelly. *COTS Product Selection for Safety-Critical Systems*. In Proceedings of the 3rd International Conference on COTS-Based Software Systems (ICCBSS'04), Springer, 2004.

[8] Fan Ye, Tom Kelly. *Criticality Analysis for COTS Software Components*. In Proceedings of the 22nd International System Safety Conference (ISSC'04), Providence, Rhode Island, USA, 2004.

[9] Thomas Tamandl, Peter Preininger. *Online Self Tests for Microcontrollers in Safety Related Systems*. 5th IEEE International Conference on Industrial Informatics, Vienna, June 2007.

[10] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, Frédéric Salles. *Dependability of COTS Microkernel-Based Systems*. IEEE Transactions on Computers, Vol.51, No.2, February 2002.

[11] CENELEC EN50126. *Railway applications – Specification and Demonstration of Reliability, Availability, Maintainability and 6060 Safety (RAMS).* Comité Européen de Normalisation Electrotechnique, 1999.

[12] CENELEC EN50128. *Railway Applications - Communication, signalling and processing systems - Software for railway control and protection systems.* Comité Européen de Normalisation Electrotechnique, 2001.

[13] CENELEC EN50129. *Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling.* Comité Européen de Normalisation Electrotechnique, 2003.

[14] POSIX Standard: IEEE Standard 1003.1-2001 and related (POSIX.1, POSIX.2, POSIX.4, POSIX.4a). Institute of Electrical and Electronic Engineers, 2001.

[15] CENELEC Standard 50159-1, Safety-related communication in closed transmission systems. Comité Européen de Normalisation Electrotechnique, 2003.

[16] Georg Hartner. Safety *Supervision Layer – Fault Detection for Operating Systems in Fail-Safe Environments*, Diploma Thesis, Institute for Computer Technologies, TU Vienna, January 2008, unpublished[4].

[3] http://www.idi.ntnu.no/emner/dt8100/Essay2007/aas-essay07.pdf
[4] http://publik.tuwien.ac.at/files/pub-et_13621.pdf