

An Architecture for a Safety-Critical Steer-by-Wire System

Dr. Juan R. Pimentel

Kettering University, Flint, MI, USA

Copyright © 2004 Society of Automotive Engineers, Inc.

ABSTRACT

A hardware and software architecture suitable for a safety-critical steer-by-wire systems is presented. The architecture supports three major failure modes and features several safety protocols and mechanisms. Failures due to component failures, software errors, and human errors are handled by the architecture and safety protocols. A test implementation using replicated communication channels, controllers, sensors, and actuators has been performed. The test implementation uses the CAN protocol, Motorola S12 microcontrollers, and Microchip MCP250XX components with a steering wheel and road wheel simulator. The focus of the paper is on the application level, using system engineering principles which incorporate a holistic approach to achieve safety at various levels.

INTRODUCTION

The increased use of microcontrollers in modern automotive systems has brought many benefits, such as merging chassis control systems to achieve active safety with passive-safety systems. For instance, the airbag controller can be combined with sensor inputs to enable new functions, such as activating passive safety features (i.e. tightening seatbelt retainers) when chassis control electronics detect an "out of control" condition. Stability control can be linked to steering or perform automatic control of oversteering. Unfortunately, it has also brought the potential for catastrophic failures [1]. Thus, the application of microcontrollers requires extreme care in order to produce a dependable system. Dependability involves reliability, safety, availability, and security but in this paper we are only concerned with safety and to a lesser extent reliability. Safety becomes important when applications include systems where the consequences of failure are serious and may involve grave danger to human life and property.

The area of system safety is well-established, and procedures exist to identify and analyze electromechanical hazards along with techniques to eliminate or limit hazards in a final product. Unfortunately, much more is known about how to engineer safe mechanical systems than safe computing systems, particularly when software is a major component of the engineered system. With the increased use of software in safety-critical components of complex systems, government agencies and other institutions are increasingly including requirements for software hazard analysis and verification of software safety (e.g., MISRA, MIL-STD-882B, IEC 61508, DO-178B). More effective modeling and analysis tools are needed to help automotive engineers perform safety analyses of systems involving electronic and information technology components.

Thus the objective of this paper is to address the design and implementation of a highly dependable software architecture for a simulated "steer-by-wire" system operating in a network environment (CAN). In addition, two dependable controllers have been developed to exercise the architecture: a simulated one and an actual one using a COTS (commercially available over the shelf) microcontrollers.

Safety Analysis.

Whereas *system reliability* deals with the issue of ensuring that a system performs a required task or mission for a specified time, *system safety* is concerned with ensuring that an *accident* or *mishap* does not occur in the process [1]. Usually, there are some failures that only cause a benign interruption of the system services while other failures cause catastrophic interruptions. The former are called *benign failures* while the latter are called *catastrophic failures*. This situation is depicted in Fig. 1

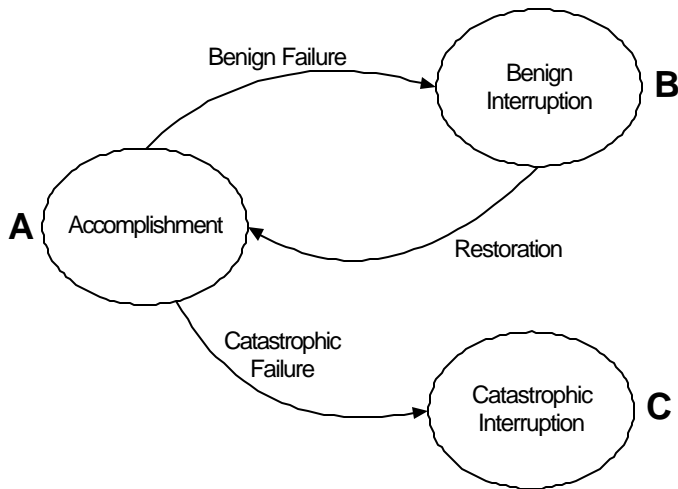


Figure 1. States relevant for the definition of safety.

Accidents or mishaps are unplanned and undesirable events or series of events that could result in injury, illness, death, or damage to or loss of property or equipment. A *hazard* is an undesirable condition that has the potential to cause or contribute to a mishap. The situation that results from the occurrence of a mishap is called a *failure mode*. Thus according to these definitions, a mishap occurs when the conditions of a hazard are fulfilled. Hazards and mishaps can be classified in various severity levels ranging from negligible to catastrophic.

More specifically, hazards can be categorized by the aggregate probability of the occurrence of the individual conditions that make up the hazard and by the seriousness of the effects of the resulting mishaps. Together these constitute risk. More specifically,

$$risk = h_s h_p$$

Where: h_s is the hazard severity and h_p is the hazard probability.

Because of the severity and probabilistic aspects of risk, there are two major views of safety: probabilistic, and system [1]. The probabilistic view is a more theoretical view that uses mathematical models (e.g., Markov Chains) to analyze safety and reliability issues. The system view is more practical and holistic. It includes all non-probabilistic issues, and is used for design and implementation rather than analysis. In this paper we are primarily concerned with system level safety. The first step in a safety analysis process is to identify the system

hazards and assess their severity and probability (i.e., risk). The vast majority of available safety tools and methods support severity analysis [2,3].

The overall goal when designing a safety-critical system is to eliminate hazards from the design, or (if that is not feasible) to minimize risk by modifying the design so that there is a very low probability of the hazard occurring. To demonstrate that a system is safe, it is first necessary to ensure that given that the specifications are correctly implemented and no failures occur, the operation of the system will not result in an accident or mishap. Second, the risk of faults or failures leading to a mishap must be eliminated or minimized by using fault-tolerant or fail-safe procedures. If it is not possible to completely eliminate hazards, then the exposure time (length of time of occurrence) of the hazard must be minimized in order to reduce risk.

One concern in this paper is handling hazards that result from component failures. Another concern involves identifying and eliminating hazards which may inherently exist in a system as a result of other failures, such as those due to design errors or unforeseen events. The following is a list of control failures that needs to be considered in any system safety analysis:

- A required event that does not occur.
- An undesirable event
- An incorrect sequence of desired events
- Two incompatible events occurring simultaneously
- Timing failures in event sequences
- Exceeding maximum time constraints between events
- Failing to ensure minimum time constraints between events

Durational failures (i.e., a condition or set of conditions fail to hold for a particular amount of time).

Safety Goals.

It is necessary to decide what qualities of safety mechanisms are important to analyze. A list of possible safety goals include recoverability, fault tolerance, and fail-safety [5]. A process is recoverable if, after the occurrence of a failure, the control of the process is not lost, and will return to normal execution in an acceptable amount of time. A process is fault tolerant if a mechanism exists so that when there are failures, the system can continue to operate, perhaps in a degraded level of performance or functionality. A system is fail-safe if no matter what combination of failures occur, they do not lead to an unsafe state.

A process for safety analysis.

Several automotive associations and institutions (e.g., MISRA) and standards (e.g., DO-178B) are concerned with guidelines, requirements, techniques, and processes to deal with hazards and safety issues. A key aspect of these guidelines is that the hazards associated with a system must be both understood and taken into consideration from the beginning of its design cycle. The following have been identified as important [13]:

- Assess the risks associated with the behavior of a system;
- Do this early enough in order to take design actions that can reduce those risks to an acceptable level;
- Provide documentary evidence of the reasoning that lies behind the design decisions made.

A number of processes for safety analysis exist or have been suggested [7] that involve the following stages: preliminary safety analysis (PSA), preliminary hazard analysis (PHA), detailed safety analysis (DSA), and safety integrity levels (SIL).

A STEER-BY-WIRE SYSTEM

As depicted in Fig. 2, a steer by wire system is composed of three main blocks: the hand (i.e., steering) wheel, the steer-by-wire controller, and the road wheels. The driver operates the hand wheel to turn the vehicle and sends the *steering angle* signal to the controller. The controller will perform some control functions associated with the vehicle's steering function and output an actuator angle for the road wheels that in turn will turn the wheels through an actuator. The feedback signals (actuator feedback and wheel feedback) involve some kind of force or torque sensors and are necessary so that the driver get the feeling of turning a traditional steering wheel and feel the effect of turning the wheels on a certain type of road. Although the system of Fig. 2 could operate in a networked or non-networked environment, we are interested in the former.

Safety-critical requirements

As noted, we are interested in a distributed steer-by-wire architecture involving the following components: ECU's, communication lines (e.g., CAN buses), and appropriate sensors and actuators. Thus the system requirements apply to the entire distributed architecture (i.e., hand wheel, road wheel, controller, ECU's, buses, software, sensors and actuators).

The overall safety-critical requirements for a given system belong to the following categories:

- Failure mode requirements
- Safety goal requirements
- Domain requirements
- Development environment requirements

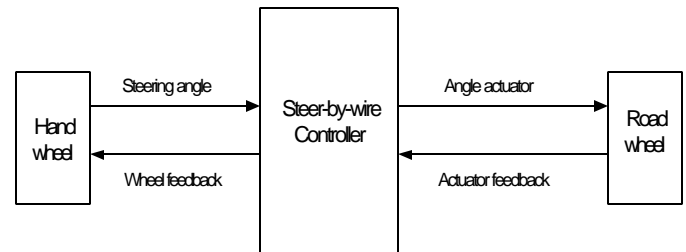


Fig. 2. Block diagram of the steer-by-wire system.

Failure mode requirements involve minimizing the probability of occurrence of a set of identified set of failure modes (see below). *Safety goal requirements* involve the use of specific procedures or protocols to ensure the safety goals stated above in terms of recoverability, fault tolerance, and fail-safety. Regarding recoverability, we require that the control of the process is not lost and is returned to normal execution in real time. For our steer-by-wire system a target recoverability real-time value is 50 ms. The fault tolerant requirement is that the system does not have a single point of failure (i.e., the system must tolerate any single fault). An additional desirable requirement is that the system can tolerate some double faults. The fail-safe requirement states that no matter what combination of failures occur in any of the components, they do not lead to an unsafe state. *Value domain requirements* involve the verification of the expected value range of certain physical variables of interest (e.g., required steering angle, actuator angle, actual road wheel angle). *Time domain requirements* involve heartbeat messages issued by the controller every 20 ms and a maximum message latency (for any message) of 20 ms. The development environment requirements are special in that they are optional, and are only needed at the developmental stage if a high degree of flexibility and modularity is needed, mostly for testing, verification, and validation purposes. For our case we require a fault generator for both, a simulation and actual environments and a generic development environment that supports several communication channels, replicated components, and flexible software development.

Failure modes

From a safety standpoint one can identify a number of hazardous states (failure modes) and focus the analysis on each of these states separately from one another. This is so because the concern is on safety rather than correctness. That is, a system is *safe* if it is free from *mishaps* even if it does not accomplish its mission or functional objectives.

Based on the drive-by-wire system, we can distinguish the following three failure modes:

1. The road-wheels do not respond to a command from the hand-wheel. (*rw-cmd*)
2. The road-wheels turn by themselves without any command from the driver. (*rw-au*)
3. There is no road feedback to the driver. (*dr-fb*)

There are many underlying reasons that would cause the above failure modes. For example, for a basic system with no replicated components, failure mode *rw-cmd* occurs when there is loss of communication, a failure in the controller, a failure in the hardware sensor, a failure in the road wheel actuator or some other failure. Likewise, failure mode *rw-au* occurs when there is a software error, a sensor or actuator failure in such a way that would cause the autonomous (i.e., by itself) turning of the road-wheel. Failure mode *dr-fb* occurs when there is a failure in any component in the feedback path from road wheel to hand wheel. Accordingly, we are interested in appropriate hardware and software architectures that, together with recoverability, fault tolerance, or fail-safe mechanisms minimize the hazard severity of a steer-by-wire when the above failure modes occur.

A SAFETY-CRITICAL HARDWARE ARCHITECTURE

The purpose of this section is to discuss architectures that will minimize the hazard severity of a steer-by-wire system upon the occurrence of any of the three failure modes discussed previously. The architectures will be complemented with recoverability, fault tolerant, or fail-safe mechanisms. Two possible distributed architectures are depicted in Figs. 3(a) and 3(b). The main feature shared by both architectures is the use of a number (two or higher) of replicated components for sensors, controllers, CAN buses, and actuators. It can be seen that in the architecture of Fig. 3(a) not all of the components are connected to the communication system (either of the two CAN buses). This is so because it is possible to connect sensor and actuators directly to the I/O port of a microcontroller. Because of the availability of

low-cost, simple sensor and actuator CAN nodes that do not require microcontrollers (e.g., Microchip MCP25050) is it a good idea to interface all sensors and actuators to the replicated CAN bus as depicted in the architecture of Fig. 3(b). The architecture used for our simulation and implementation studies is that of Fig. 3(b). Table 1 summarizes the various devices of the distributed hardware architecture.

Device	Description
CAN1, CAN2	Primary and redundant (secondary, backup) CAN bus respectively.
ECU1, ECU2	Primary and redundant ECU. They execute the main automotive application function.
MC3, MC4	Primary and redundant microcontrollers. They handle sensors and actuators signals at the road wheel.
S1-a, S1-b	Primary and redundant steering angle sensor at the hand-wheel
S2-a, S2-b	Primary and redundant feedback torque sensor at the road wheel
A1-a, A1-b	Primary and redundant feedback torque actuator at the hand-wheel
A2-a, A2-b	Primary and redundant steering angle actuator at the road wheel

Table 1. Device summary of the distributed hardware architecture

Fig. 3. CAN network where message priorities are defined by nodes.

As noted, both architectures includes duplicated sensors and actuators at the hand wheel and road wheel, duplicated controllers (ECU1 and ECU2), duplicated microcontrollers (MC3 and MC4) for sensors and actuators, and duplicated CAN buses (CAN1 and CAN2). This massive replication helps meet the fault tolerant requirement because the system does not have a single point of failure. Software plays a fundamental role to implement schemes to deal with other recoverability, fault tolerant, and fail-safe requirements. In addition to the application functionality to be implemented by software, communication and physical interfacing are important considerations. Thus there are several possibilities for appropriate safety-critical hardware architectures. Options include networked or direct I/O, microcontroller-based or special CAN interfaces for sensors and actuators, and degree of replicated components.

To meet real-time requirements, the controller software is designed on top of a real-time basic software executive as shown in Fig. 4. This arrangement will work provided that each software module finishes execution before the next software

module and CAN message latency is bounded by the basic cycle period (depending on the message scheduling for the system in question). For our case, the basic cycle period is dictated by the heart beat message, and it is 20 ms. Because of the key role of software for meeting the requirements a safety-critical architecture, we put all the software

communication lines (CAN1 and CAN2). Let us consider the architecture of Fig. 3(b) where primary and secondary (i.e., backup or redundant) sensors, actuators, and controllers are directly connected to the dual-bus system.

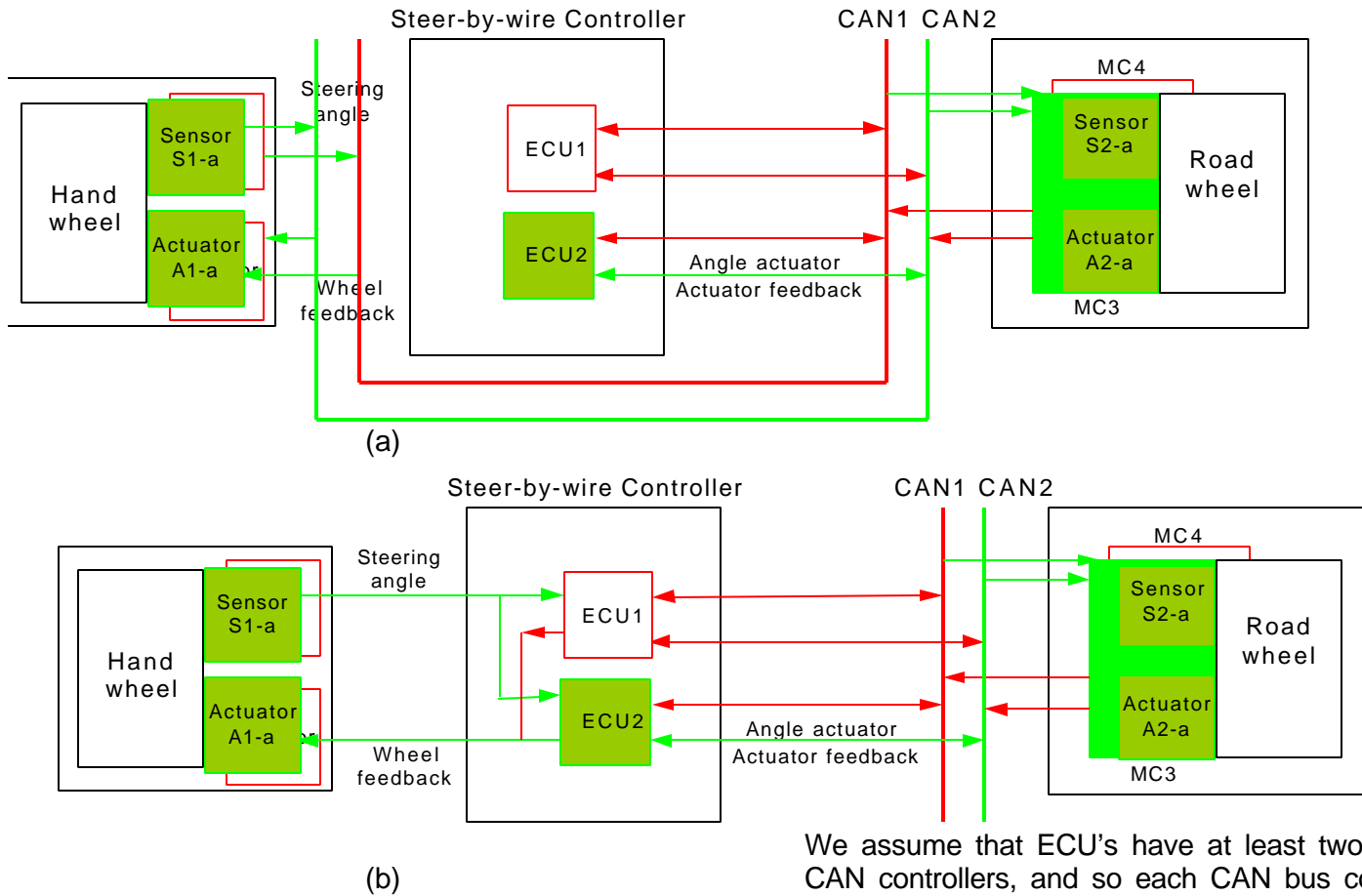


Fig. 3. Hardware architectures for a dependable, steer-by-wire system.

safety mechanisms in specific software modules called *safeware*, as depicted in Fig. 5. The architecture distinguishes between safeware modules for input/output, application, and communication (intra- and inter-node) functions. It is important to point out that safeware deals explicitly with safety rather than functional issues. This fact will help understand the generic safeware software architecture of Fig. 6 and will be explained below.

Description of the steer-by-wire hardware architecture

As shown in Fig. 3, the hardware architecture is made up of a hand-wheel unit, a road wheel unit, the steer-by-wire controller, and the redundant CAN

We assume that ECU's have at least two built-in CAN controllers, and so each CAN bus controller can be treated as a single "line replaceable unit" (LRU), (i.e., a failure of the CAN bus controller is independent of the failure of other CAN bus controllers). However, the failure of a CAN bus controller is correlated to the failure of the ECU (e.g., an ECU failure will cause the failure of all CAN controllers supported by that particular ECU). We further assume that there are redundant sensors and actuators at the road-wheel unit but unlike those at the hand wheel, they are connected to the CAN bus via a microcontroller¹. Thus it is implied that there are two redundant microcontrollers at the road wheel unit handling the sensing and actuating functions. The primary assumption in any redundant configuration is that when the operational (i.e.,

¹ Actually the choice of using a microcontroller or direct sensor/actuator interface to a CAN bus depend on the goals of a project. For experimental and evaluation types of projects, the use of a microcontroller might be advantageous, whereas for actual use in a vehicle (i.e., target implementation) the low cost direct interface is preferred.

primary) component of a redundant configuration fails, another component (i.e., secondary or redundant) will take over the function of the failed component. It is important to realize that the designation of primary or secondary is dynamic rather than fixed or static. A component may change from primary to secondary and back to primary according to the details of an error detection and recovery mechanism.

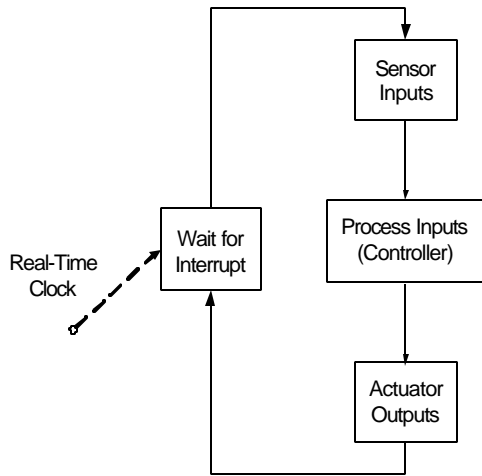


Fig. 4. A real-time software executive

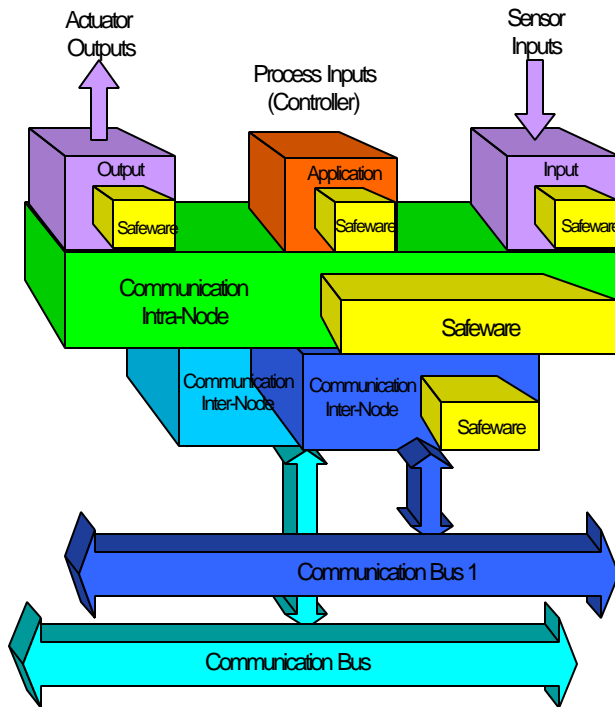


Figure 5. Major software components of the software architecture.

Assuming that at any given instant the primary components are ECU2, CAN2, S1-a, S2-a, A1-a, and A2-

a, the system operates as follows: when a driver command is sent from the steering wheel, the primary sensor in the hand wheel (S1-a) will send the corresponding signal to ECU2 on both CAN channels. The components actually keep track of which CAN bus is the primary one according to an error detection and recovery mechanism [12] not detailed here. Only ECU2 will perform its functions (calculations) and output its messages on both CAN buses. Meanwhile ECU1 will monitor ECU2 to detect failures and will take over (switched to primary) accordingly. The primary road wheel actuator (A2-a) will read from the primary (CAN2) bus to generate signals for the road-wheel motor. Likewise, the primary road-wheel sensor (S2-a) will send actuator feedback information to ECU2 using both CAN buses. Finally, ECU2 will forward the information to the primary actuator (A1-a) at the hand-wheel on both CAN buses.

A Safety-Critical Software architecture

The hardware architectures of Fig. 3, together with the controller software organization of Fig. 4, already determine some major features of the safety-critical software architecture. There are some additional desirable features for the software architecture such as: well-defined interfaces, independent of actual applications, independent of the degree of safety offered (e.g., number of redundant components and error recovery methods), independent of communication protocols, and independent of microcontroller type. Thus, the software architecture must be generic and hierarchical so that at each level of the hierarchy, additional details can be added to result in a specific software architecture.

The following are the main functions to be implemented by software components [9]:

1. Application-specific
2. Error and failure detection (local and global)
3. Damage assessment and confinement
4. Error recovery
5. Fault treatment
6. Hardware reconfiguration
7. Software reconfiguration
8. Recovery strategy (local and global)

Thus, the specification and description of the software architecture must be hierarchical, starting with generic requirements and progressing to more detailed and specific safety-related protocols and mechanisms. Fig. 6 depicts the highest level software architecture that meets the requirements outlined above. This architecture has been adapted

from a similar architecture to deal with availability issues of distributed embedded systems [11]. This software architecture can be used any time there is an independent and major piece of software that deals specifically with safety functions, for example ECU1, ECU2, MC3, or MC4 in Fig. 3. The software architecture emphasizes the various safety functions listed above, such as error and failure detection, reconfiguration, and recovery strategies, both at the hardware and software level.

The model of Fig. 6 assumes a primary version (on the left side) together with a secondary (i.e., backup) version (on the right side) of a generic block labeled the Nth block. There are also two block components, a hardware (on the top) and a software (on the bottom). The notation in the model is compatible with the standard block diagram notation of systems engineering. It is also compatible with the Petri Net notation because it can be used to depict the more detailed behavior at other levels of the software hierarchy. Petri Nets have been extensively used for probabilistic safety analysis [6,9,10,11]. The function of the N-SRec component is to reconfigure the software of the Nth block after failure detection. Likewise, the function of the N-HRec component is to reconfigure the hardware of the Nth block. N-Strat is a global recovery strategy that takes into account detailed hardware and software status to determine future system actions through software. N-Stop is a system shutdown action taken when there is a fault that can no longer be tolerated (covered). A summary of the functions of the various block components is given in Table 2.

Name	Function
N-Prop	Propagation of hardware error to the hosted software
N-Stop	Software stop after activation of a permanent fault in the hosting hardware
N-Hprop	Propagation (to other blocks) of hardware error
N-Sprop	Propagation (to other blocks) of software error
N-Hrec	Hardware reconfiguration and repair
N-Srec	Software reconfiguration and repair
N-Strat	Global recovery strategy

Table 2. A summary of the various block components.

Safety protocols and mechanisms

The model in Fig. 6 provides an appropriate framework for defining several safety protocols and mechanisms, particularly those dealing with reconfiguration after failure detection. In particular, a flexible error detection and reconfiguration protocol

has been developed and implemented as part of the proposed architecture [12]. The reconfiguration protocol supports any number of replicated hardware and software components that together select a primary component. The remaining components act as secondary, tertiary, components, depending on their number. With the exception of communication buses, the set of redundant components behaves just like one primary component. The main advantage of the protocol is that software complexity is reduced by dealing with just one primary component rather than explicitly dealing with the primary and all redundant components in an explicit fashion.

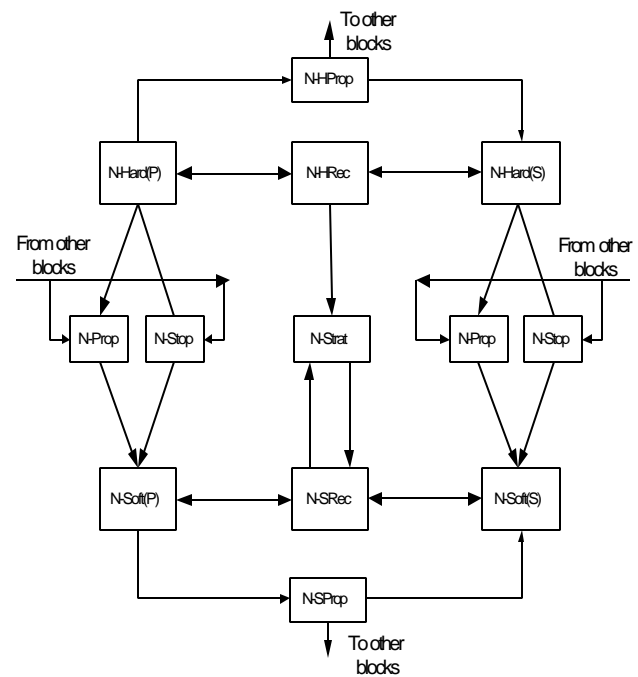


Fig. 4. Generic safety software (safeware) block architecture.

Implementation and evaluation

The hardware and software architectures have been implemented in two environments, a simulated one and an actual one using the CAN protocol as the implementation network. The simulated environment was built using CANoe. We are currently implementing an actual system using Motorola S12 microcontrollers as ECU's and Microchip MCP25020 and MCP25050 analog nodes with CAN interface for all sensors and actuators. To have more flexibility with the initial implementation, we have developed a steering wheel and road wheel simulator, also using CANoe. We are currently porting the actual hardware and software

implementation to an actual golf cart vehicle for final experimentation and demonstration.

We have tested and verified all components of the software architecture shown in Fig. 6 in the CANoe simulator of the entire system according to the safety-critical requirements stated above. In particular the N-Hprop, N-Hrec, N-Strat, N-Srec and N-Sprop functionality was thoroughly tested, as they are part of the error detection and recovery protocol [12]. As to the failure mode requirements, we tested to see if the system minimizes the occurrence and severity of the failure modes by fault injection (e.g., a fault in a hand wheel sensor for testing failure mode *rw-cmd*). We tested the recoverability safety goal by ensuring that the system returned to stable operation after the occurrence of faults. The fault tolerance safety goal was tested by making sure there was not a single point of failure and that the system tolerated some double and triple faults (e.g., failure of one ECU and one CAN bus simultaneously). As to some fail-safe features, we tested a mechanism where the road wheels return to the straight position (at a slow rate) whenever there is no command from the hand wheel or when a failure is detected in the hand wheel sensor. The time domain requirement was tested by ensuring a steady pattern of heartbeat messages and verifying message latencies for all messages. Value domain requirements were tested by verifying that the actuator angles were within valid ranges. The distributed embedded systems laboratory at Kettering University with 6 seats of CANoe and the IAR development systems proved to be an appropriate development environment for this project. We plan to use similar system engineering techniques to perform a thorough testing, verification, and validation of the actual system involving the golf cart vehicle when it is completed.

Discussion

This paper's main contribution is to analyze, design, simulate and implement a safety-critical steer-by-wire by focusing on the application level and using system engineering principles rather than focusing on the detailed safety-critical mechanisms provided by a communication protocol [14]. Another contribution is the holistic approach to dependability where safety is achieved at various levels: hardware and software architecture, protocol, time-triggered events (heartbeat), embedded system hardware, redundancy, error detection and fault recovery protocols, and systems engineering. The CAN protocol has some dependable features, but these are not considered sufficient for safety-critical applications unless they are supplemented by higher layer mechanisms.

SUMMARY AND CONCLUSIONS

There are several possibilities for hardware and software architectures depending upon the level of redundancy and interconnection to the communication system, and degree of error detection and fault recovery offered by the system. This paper has presented two specific architectures, and one was chosen for analysis, design, simulation, and implementation purposes. The entire system was simulated in CANoe and tested according to the requirements. The results were encouraging. The system is currently being implemented on an actual golf cart vehicle. The focus of the paper has been on the application level using system engineering principles where a holistic approach was used to achieve safety at various levels.

Acknowledgements

The author wishes to thank all students from the Spring 2003 class "CE senior capstone design" for their work implementing the CANoe simulations. In particular, J. Kaniarz, J.D. Wiener, S. T. Lim, and J. Gallagher participated in the design of the fault management protocol.

REFERENCES

- [1] Leveson, N.G., *Safeware*, Addison-Wesley.
- [2] Hammer, W., *Handbook of System and Product Safety*, Prentice-Hall, 1972.
- [3] Vesely, W.E., Goldberg, F.F., Roberts, N.H., and Haasl, D.F., *Fault Tree Handbook*, NUREG-0492, U.S. Nuclear Regulatory Commission, Jan. 1981.
- [4] Arlat, J., and J.C. Laprie, On the Dependability Evaluation of High Safety Systems, 15th IEEE Int. Symp. On Fault Tolerant Computing, pp. 318-323, June 1985, Ann Arbor, Mi., USA.
- [5] Levenson, N., and Stolzy, J.L., *Safety Analysis Using Petri Nets*, 15th IEEE Int. Symp. On Fault Tolerant Computing, pp. 358-363, June 1985, Ann Arbor, Mi., USA.
- [6] Ziegler, C., *Surete de Fonctionnement D'Architectures Informatiques Embarquees sur Automobile*, Rapport LAAS No. 96289, Toulouse, France.

[7] Amberkar, S., D'Ambrosio, J.G., Murray, B.T., Wysocki, J., and Czerny, B.J., A System-Safety Process for By-Wire Automotive Systems, SAE Congress paper, 2002.

[8] Pimentel, J.R., and Sacristan, T., A Fault Management Protocol for TTP/C, Proc. IECON'01, pp. 1800-1805, Denver, CO., Nov. 2001.

[9] Pimentel, J.R., and Salazar, M., Dependability of Distributed Control System Fault Tolerant Units, Proc. IECON'02, Seville, Spain., Nov. 2002.

[10] Choi, C.Y., Johnson, B.W., and Profeta, J.A. III, Safety Issues in the Comparative Analysis of Dependable Architectures, IEEE Trans. On Reliability, Vol 46, No.3, pp. 316-322, Sept. 1997.

[11] Kanoun, K., and Ortalo-Borrel, M., Fault-Tolerant System Dependability – Explicit Modeling of Hardware and Software Component – Interactions, , IEEE Trans. On Reliability, Vol 49, No.4, pp. 363-376, Dec. 2000.

[12] Pimentel, J.R., and Kaniarz, J., “A CAN-based Application Level Error Detection and Fault Management Protocol”, in preparation, ECE Department, Kettering University, Flint, Michigan, Oct. 2003.

[13] Jetsy, P.H., Hoble, K.M., Evans, R., and Kendall, I., Safety Analysis of Vehicle-Based Systems, European conference, 2000.

[14] Kopetz, H. “Fault Containment and Error Detection in the Time-Triggered Architecture,” Proc. IEEE Int. Symp. On Autonomous Decentralized Systems, ISADS 2003, pp. 139-148, April 2003.

Email: jpimente@kettering.edu

ACRONYMS, ABBREVIATIONS

CAN: controller area network

COTS: commercially over the shelf components

PSA: primary safety analysis

DSA: detailed safety analysis

SIL: safety integrity level

ECU: electronic control unit

I/O: input/output

MC: microcontroller

LRU: line replaceable unit

CONTACT

Dr. Juan Pimentel is a Professor of Computer Engineering at Kettering University. He holds a Ph.D. degree in Electrical Engineering from the University of Virginia. Dr. Pimentel has done extensive research in the U.S, Germany, Spain, and Colombia. He is a Fulbright Scholar and an associate editor of the IEEE Transactions on Industrial Electronics and the Transactions on Mobile Computing. His main research areas are: distributed embedded systems, real-time networks and protocols, dependable systems, and electric propulsion systems. He is a member of Tau beta pi, Eta kappa nu, Sigma xi, IEEE, and SAE.