Fault-Tolerance of Embedded Systems with Automotive Applications

Ratnesh Kumar, Dept. of Elec. & Comp. Eng., Iowa State Univ. (ISU)
Shengbing Jiang, General Motors R&D and Planning, Warren, MI

Introduction

Motivated by automative applications, we propose an approach for fault-tolerance in embedded systems that is based on fault detection/prediction, identification and recovery. Computing (for controls, monitoring, other services) in large systems, including safety-critical systems such as automobiles, aircrafts, nuclear plants, medical devices, etc., is performed using distributed embedded systems. There are many documented cases of failures of such systems due to software/hardware errors. The existing simulation/testing/verification practices cannot guarantee that a deployed embedded system will continue to operate error-free. (The problem is in general undecidable.) So it is important that measures be built-in for providing tolerance against any design-errors or runtime-faults that can compromise the safety of the users or the surrounding environment.

Modern vehicles will be equipped with advanced features such as collision avoidance, adaptive cruise control, lane centering/changing, all of which will be implemented in software running over distributed embedded systems, typically interconnected using a CAN bus. The existing approaches to fault-tolerance are either restrictive or not cost effective. E.g., design-diversity based approaches provide tolerance at the cost of redundancy and do not necessarily identify the root-cause of failures. On the other hand, the execption handlers based approaches can only react to faults that have been anticipated in advance whereas certain faults, specially those of software, are hard to anticipate in advance. Similarly the simplex architecture is limited by the ability to accurately define the safety envelope for its highly reliable mode of operation. New approaches are needed for fault tolerance of safety critical automotive applications. We envision introducing a new module to the exisiting vehicle control architecture to enable software fault-tolerance as shown in Figure 1. A similar architecure will be employed for fault-tolerance in hardware components—We are targetting the CAN bus as a concrete benchmark application.

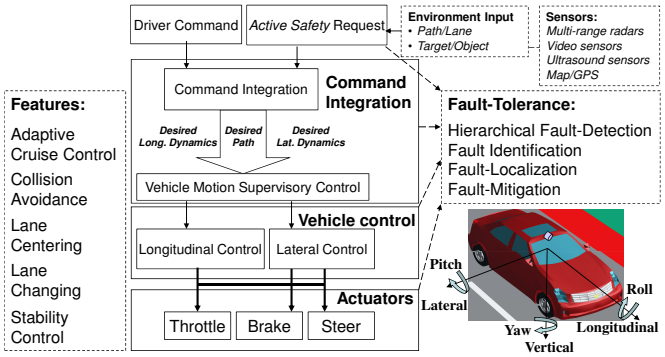

Figure 1: Control Architecture for software fault tolerance

Our Approach

Our approach to software fault-tolerance (which we plan to adapt for hardware fault-tolerance as well) consists of (i) Hierarchical two-tier monitoring of software and system behaviors against their specfications for fault-detection, (ii) Model-based fault-diagnosis to identify the faulty component in case a fault is detected at the system level, (iii) Model-based analysis of run-time data for fault-localization to identify and debug the faulty lines of code, and (iv) On-line reconfiguration for fault-mitigation and recovery. The approach assumes the availability of the following information:

1. Software (component) level specs to detect software (component) faults.
2. System (= software + computing-platform + plant) level specs to detect system faults, in case missed at component level owing to incompleteness of compnent level specs.
3. Nonfaulty plant model to enable ruling out plant fault when a system fault is detected (a residue-based analysis is used for this purpose).
4. Sensor noise distributions to determine appropriate cut-offs for system-level monitors and residue generators in order to "balance out" false-alarm/missed-detection/detection-delay.
5. Computing-platform specs and tests, and also a dual-redundancy computing to enable ruling out transient/permanent computing-platform fault when a system fault is detected.

6. Existence of simplex architecture or other alternative to enable mitigation/recovery (a system level analysis will be used to determine the control-switching policy to attain recovery).
7. Software/hardware-task execution times (including monitoring overhead), their periods and the task dependency graph (to detect any timing fault caused by added task of monitoring).

Note our approach does not require the availability of any software (component) model for monitoring (ie., only their specifications are needed). In case software (component) models are available, more could be done, eg., fault-localization to aid debugging. We use input/output extended finite automata (I/O-EFA) for software (component) as well as system level modeling, and also for modeling the specification monitors. (Since the control changes only at the discrete times when the system/environment states are sampled, the controlled system has a discrete-time hybrid dynamics which can be modeled as I/O-EFA.) The approach for software fault-tolerance (which we will adapt for hardware fault-tolerance) is illustrated in Figure 2 and involves the following steps.

Hierarchical Fault-Detection. The observed input/output behavior of software components and also the entire system is monitored against their specifications. A software failure is immediately detected when an observed behavior is rejected by a software level monitor, whereas when a system level fault is detected further diagnosis is performed to rule out the faults of plant or of computing-platform.

Fault-Diagnosis or Identification. To rule out a fault in the plant hardware, the residue method is used, where residue is defined as the errot between the observed and model-predicted outputs of the plant. The residue is small if and only if there is no hardware fault. Note that residue generation requires the model of non-faulty plant hardware as well as noise distribution parameters. Computing-platform hardware faults can be of two types: Transient and permanent. For ruling out transient faults, redundant computing-platform computation and comparison is used. For ruling out any permanent fault, tests such as challenge-response are are triggered and executed.

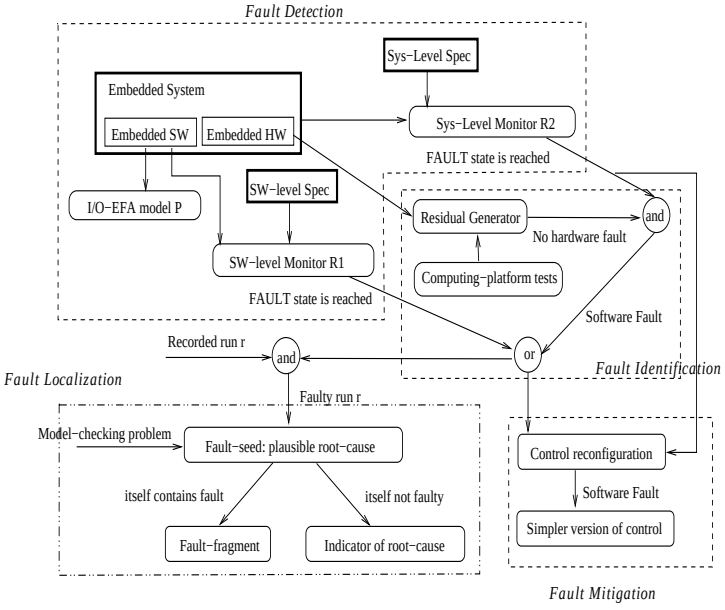

Figure 2: Approach for fault tolerance of embedded software

Software Fault-Localization. This involves localizing the root-cause, namely locating the faulty lines of code, or the indicators for missing lines of code. We formalize the concept of a plausible root-cause by introducing the notion of a fault-seed: A subset of statements included in a faulty-run is called a fault-seed if their influencive execution in *any* run of the software causes a failure to occur. Note a fault-seed can itself contain faults in which case itself a root-cause and referred as fault-fragment or, if not itself faulty, its execution is essential for the manifestation of failure caused by some missing code and in this case is an indicator of the root-cause. Thus the approach is helpful in localizing faulty lines of code or an indicator for missing lines of code (as the case may be). A fault-seed can be algorithmically computed: Checking whether a chosen subset of statements is a fault-seed is formulated as an instance of a model-checking problem.

Reconfiguration for Fault-Mitigation. Once a controller fault is detected, measures are taken to reconfigure the control to ensure the safety and stability of the overall system. For this, reconfiguration strategies are being developed in the framework of hybrid systems. If a software fault is detected, control is switched to simpler version that is known to be reliable or to an alternative. For example if braking software has fault, we can switch to either a simpler reliable braking software or, steering may be used to achieve braking (by forcing the wheels inwards towards each-other).

Application to CAN Bus

Controller area networks (CAN) are widely used in automotive embedded control systems for communications among multiple ECUs (electronic control units). The CAN communication protocol is a CSMA/CD (Carrier Sense Multiple Access / Collision Detection) protocol. In CAN protocol, a nondestructive bitwise arbitration method is utilized, i.e., the highest priority message remains intact after arbitration is completed even if collisions are detected. CAN is a message-based broadcast bus. All nodes in the system receive every message transmitted on the bus, ie., it is up to each node in the system to decide whether the message received should be immediately discarded or kept to be processed. A CAN communication from a sender to a receiver involves multiple hardware devices and different software components. At the high level, the communication involves the sender ECU node, the bus, and the receiver ECU node. At the low level, in the sender and receiver ECU nodes, the communication involves hardware (like CPU, memory, etc.) and software (like application software, Operating System (OS) scheduler, communication software stack, etc.); on the bus, it involves bus controllers, transceivers, connectors and cables.

It is very important to detect, diagnose, and mitigate failures in CAN systems accurately and timely. In the communication, different types of failures (transient, intermittent and permanent) could happen in different sofware/hardware components. Although we could detect some failures at some layers using different approaches (like the CAN physical layer faults could be detected at the CAN data link layer by using CRC, bit stuffing, acknowledgement, etc.; and the loss of periodic messages could be detected at the receiver side by using time-out), we still need an integrated approach for fault detection and isolation in CAN systems so that we could pinpoint the exact faulty device/component. At the same time we want to minimize the resources (CPU and memory) needed for the fault detection and isolation while maximize the fault coverage and the accuracy of the fault isolation. We plan to extend our hierarchical approach to solve the above problem.

Author Biographies

Dr. Ratnesh Kumar is a Professor of ECE at the Iowa State University since 2002. Prior to this, he held faculty position at the Univ. of Kentucky (1991-2002) in ECE, and has held visiting positions in academia, research labs as well as industry: University of Maryland, College Park, Applied Research Laboratory (at Penn State Univ.), NASA Ames, Idaho National Laboratory, and United Technologies Research Center. He received B.Tech. in EE from IIT Kanpur in 1987, and M.S. and Ph.D. in ECE from the Univ. of Texas, Austin in 1989 and 1991, respectively.

His research interests are in modeling, verification, testing, control, monitoring, diagnosis, and prognosis of event-driven, real-time hybrid systems, and their applications to embedded control systems and software, cyberphysical systems, web-services, autonomous systems, sensor networks, power systems, and precision farming. He serves on the program committee for the IEEE Control Systems Society, the International Workshop on Discrete Event Systems, the International Workshop on Dependable Control of Discrete Systems, and the IEEE Workshop on Software Cybernetics. He is or has been an associate editor of SIAM Journal on Control and Optimization, IEEE Transactions on Robotics and Automation, Journal of Discrete Event Dynamical Systems, International Journal on Discrete Event Control Systems, and IEEE Control Systems Society. He has been General Co-Chair for the International Workshop on Discrete Event Systems, and a Program Co-Chair for the IEEE Workshop on Software Cybernetics. He is a Fellow of the IEEE.

Dr. Shengbing Jiang is employed with General Motors R&D, Warren, MI, since 2002, where he holds the position as a staff researcher in the vehicle health management group of Electrical & Controls Integration Lab. His research interests include formal verification, supervisory control, and failure diagnosis of discrete event and hybrid systems, and their applications in automotive embedded control systems. He has extensive background in processes and techniques for developing automotive control systems. He received the B.S. degree in electrical engineering from the University of Science and Technology of China, Hefei, China, in 1987, the M.S. degree in electrical engineering from East China Institute of Technology, Nanjing, China, in 1990, and the Ph.D. degree in electrical engineering from the University of Kentucky, Lexington, in 2002.