Out-of-Norm Assertions

P. Peti, R. Obermaisser, H. Kopetz
Vienna University of Technology
Austria
email: {php,ro,hk}@vmars.tuwien.ac.at

***Abstract*— The increasing use of electronics in transport systems, such as the automotive and avionic domain, has lead to dramatic improvements with respect to functionality, safety, and cost. However, with this growth of electronics the likelihood of failures due to faults originating from electronic equipment also increases. Although permanent failure rates are constantly diminishing due to improvements in manufacturing, the downsizing of semiconductor features has lead to a significant increase in transient system disturbances. Furthermore, transients are frequently the precursors of upcoming permanent failures. In order to cope with this development, a diagnostic subsystem must especially be designed to detect and analyze such transients to reduce the failure-not-found ratio in today's systems. Therefore, diagnostic detection mechanisms must be devised that refrain from traditional error detection techniques operating only on component-local data in favor of a system-wide view to detect and analyze correlated failures and infer the corresponding fault.**

In this work, we present Out-of-Norm Assertions (ONAs) as a diagnostic mechanisms operating on the distributed state to detect correlated component malfunction. ONAs take the characteristics of faults in the time, value and space domain into account in order to discriminate between different types of faults that are affecting the operation of the distributed system. Since ONAs are specified on the interface state mutual error detection of interface state variables is performed. In contrast to bivalent assertions that need to indisputably decide on correct or incorrect system states at the time of occurrence, the proposed ONAs are also useful in the detection of system irregularities that cannot be forced into the predominant bivalent assessment scheme.

I. Introduction

There is a significant trend in the automotive and avionics industry to increase the number of electronic devices to provide a functionality that goes beyond common mechanic/hydraulic systems. The reduction of cost, increased safety and reliability, advanced maintenance possibilities, reduced complexity, and enhanced quality of control are among the primary objectives of replacing conventional subsystems with electronic ones. For example, in the automotive domain it is estimated that more than 80% of all innovations now stem from electronics [1]. This numbers are underpinned by recent studies of the automotive electronics market that predict impressive growth rates in the next years [2].

Despite all the benefits it is important to state that with the increasing use of electronic devices the likelihood of malfunctions and thus the numbers of defective electronic components will also increase.

Many deployed On-Board Diagnosis (OBD) systems analyze the internal state of a component (e.g., plausibility checks) by applying embedded assertions in the application software in order to identify component errors. Assertions are a powerful and accepted mechanism in helping in the detection of application errors. However, such assertions operate in general only on the internal state of components. The inability to trace correlated failures of the nodes of a distributed system makes diagnosis prone to misjudgement about transient faults affecting the system. These so-called *cannot duplicate failures* frequently results in the replacement of operational Field Replaceable Units (FRUs) [3], [4]. Even worse, in many cases the faulty FRU causing the failure remains in the systems. As a consequence, these spurious failures have a lasting effect on the customer's trust in the product and the reputation of the manufacturer [5].

In this paper we introduce *Out-of-Norm Assertions (ONAs)* as a diagnostic mechanism operating on the distributed state induced by a sparse time base. The notion of *out-of-norm* originally comes from production machinery where early warnings of system breakdown enable preventative maintenance, thereby achieving enormous cost savings [6]. In this paper we extend this concept to the domain of computer systems by providing a generic mechanism that correlate anomalies, i.e. suspicious states, in the value, time, and space domain in order to reach a judgement about the correctness of a component.

The paper is structured as follows. Section II discusses related work in the domain of error and anomaly detection. Section III elaborates on the concept of distributed and interface state, in particular with respect to diagnosis. In addition, consequences of faults on the distributed state are discussed. In Section IV the concept of ONAs is introduces and relates to the concept of state. The paper concludes with a description of the application of ONAs for solving prevalent diagnostic problems in Section V.

II. Related Work

The following section elaborates on related work on error and anomaly detection.

A. Error Detection using Assertions

Assertions are boolean expressions that test the state of an executing program. Typically assertions have a form similar to

`if not` ASSERTION `then` ERROR

where ASSERTION is a predicate on the program state and ERROR indicates that an error condition is triggered.

According to [7] assertions were first proposed by Touring and rediscovered by Floyd in order to assign meanings to

programs [8]. This lead consequently to the use of assertion for formal reasoning about the correctness of sequential programs [9]. Assertions provide a logical basis for proofs of the properties of a program. Hoare introduced the so-called Hoare triple $P\{Q\}R$ to make general assertions about the values of relevant state variables. The precondition P expresses the values before, while the postcondition R expresses the values after the execution of program Q. The final assertion R implies the desired property. More information on reasoning for sequential programs using the Hoare logic can be found in [10].

In addition to verification, assertions are also a powerful and acknowledged mechanism to improve the reliability of software [11]. So-called *executable assertions* are embedded into the code and provide automatic runtime detection of errors [7], [12], [13].

B. Anomaly Detection

The phenomenon of system anomalies is subject to intensive research in many domains. This section gives a brief overview about prevalent definitions of the concept of anomaly.

Brotherton et al. [14, p. 3113] define anomalies in the context of aerospace systems as *off-nominal operations that have never been anticipated nor ever encountered before*.

In the context of embedded systems Maxion et al. [15] define the task of an anomaly detector to quantitatively decide on the distance between normal and abnormal behavior of a component based on a similarity or distance metric. The actual interpretation of the reference behavior and deployed similarity metric are usually application specific.

Anomaly detection is often based on a specific application model (e.g., jet engine), in order to compute the difference between variable values and the model-estimated outputs. The results are used to determine whether these differences indicate an anomaly compared to nominal test case [16].

During spacecraft missions each occurring system anomaly is reported for further analysis and assessment. A so-called ISA (Incident/Surprise/Anomaly) report is filled out by the operator at mission control at the time of occurrence. Later this report is extended by the analysis and concluded by the description of a possible corrective action. In contrast to a defect report, an ISA is written whenever the *behavior of the system differs from the expected (i.e. required) behavior*. The importance of this ISA lies in the provision of valuable information to the requirements engineer by capturing the gaps between the specified and implemented requirements and the user's expectation [17].

Comparative Evaluation

Assertions are a powerful and accepted mechanism facilitating in the detection of value errors. In particular, assertions are used in software engineering to ensure correctness and safety requirements. Recent studies revealed that the detection coverage of executable assertions is fairly high [18].

However, a prerequisite for the deployment of bivalent assertions is the ability to indisputably demarcate between correct and incorrect system states at the time of occurrence. System anomalies can be described in terms of a deviation from the expected nominal system behavior. However, the construction of a similarity metric is a complex task and is based on profound knowledge about the application. In many application domains, it is difficult to devise metrics and thresholds. For this reason it is necessary to devise a classification scheme that refrains from the traditional correct/incorrect classification. This diagnostic strategy allows to cope with system anomalies that cannot be judged as being correct or incorrect at the time of occurrence.

C. Automotive Diagnosis

As illustrated in Figure 1 each Electronic Control Unit (ECU) deployed in a car typically has a diagnostic subsystem that analyzes the functionality of the constituting parts (e.g., via Built-In Self Test (BIST)) or performs application specific plausibility checks, i.e. assertions, to detect errors. Once the OBD system of the car detects a violation of the

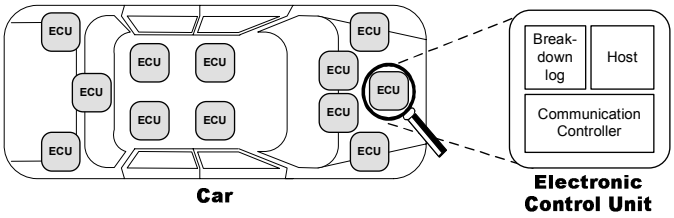

Fig. 1. Diagnostic Infrastructure of a Car

specification of an ECU, a breakdown log entry is written, and in case of a high severity, the driver is informed via the Malfunction Indicator Light (MIL). In case of an error, current diagnostic systems provide a so called *freeze frame* function, that records the condition of the vehicle when a failure occurs. The freeze frame provides important information for the failure cause analysis. The breakdown-log typically stores data on the type of fault, the state of the system, the priority, the environmental conditions, a timestamp, and information on the milage of the car. The maintenance engineer can use this collected data for getting insight into the context of the system malfunction. However, this information is often insufficient to understand the complex processes within the system that caused the system to fail. Depending on the type of inspection (e.g., garage, factory inspection, development) different parts of the breakdown log entry are analyzed.

In maintenance mode the ECUs are accessed using dedicated protocols like ISO-9141, J1850 or the Controller Area Network (CAN) based Keyword Protocol (KWP) 2000 [19], [20]. At the service station the mechanic uses a diagnostic testing device (e.g., VAG tester) to receive information about pending problems. Since most mechanics are no specialists in automotive electronics, the service technician depends on the accuracy of the diagnostic information provided by the OBD (see Figure 2). Based on this information the mechanics must be able to determine which part of the system caused

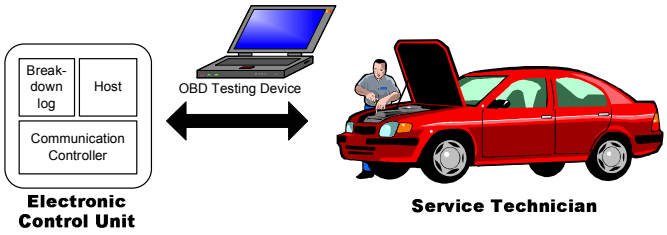

Fig. 2. Diagnosis at the Service Station

the failure and whether a replacement restores the intended functionality.

III. Distributed State

This section discusses the notion of state in distributed systems and the consequences of faults on the distributed state. We start by elaborating on the component state and express the role of the interface state as the part of the component state that is visible to all other components of the system. Furthermore, we exploit the concept of the sparse time base in order to reach a consistent distributed state. Finally, we introduce the concept of fault patterns on the distributed state to model and analyze the effects of faults.

Informally speaking, the notion of state is introduced in order to separate the past from the future (i.e. a decoupling). *The idea is that if one knows what state the system is in, he could with assurance ascertain what the output will be* [21, p. 45]. Hence, the state accumulates all past history of the given system. Apparently, this definition of state by Mesarovic and Takahara is only meaningful, if the notion of past and future (time) is relevant for the considered system. Central to this definition is the inseparable nature of time and state. The remainder of the paper is based on this definition of state.

A. Component State

A strategy for managing the complexity of technical systems is the division of complex systems into manageable subsystems. This intuitive approach to manage complexity can be found in many engineering disciplines, where large systems are assembled from prefabricated components with known and validated properties. A component is regarded as a self-contained subsystem that can be used as a building block in the design of a larger system.

In the paper we define a component as a *self-contained composite hardware/softwares subsystem that can be used as a building block in the design of a larger system* [22]. Thus, a component is considered to be a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system), which interacts with its environment by exchanging messages across linking interfaces.

This definition of a component, i.e. the inseparability of the software from its executing hardware, provides the ability to observe and describe the behavior of the component in relation to physical time. A component possesses a physical clock, the ticks of which trigger the progression from one processing step to the successive one. As a result of each processing step, a component will update deterministically its internal variables – based on the program code, the inputs, and the values of the internal variables before the processing step. We denote this component as *time-aware*, since the underlying conceptual model of a component incorporates the notion of physical time.

At startup, a component has not processed any inputs and the values of the internal variables of the component equal the initialization values. Consequently, at this point in time, the future outputs of a deterministic component depend solely on the program code and the initialization values. These static data structures of a component are usually denoted as the *initialization state*. The *history state*, on the other hand, incorporates the dynamic data structures that change over time [23]. If we do not wish to perform this differentiation, we simply speak of the *state* of the component at a particular point in time.

In distributed computer systems, the components interact by the exchange of messages across service interfaces to realize emerging services. Based on the analysis of the interactions between a component and its environment a strict separation of concerns at the interface level [24] helps to reduce complexity by enforcing a more structured design. A service interface that is provided to link components together is called a Linking Interface (LIF) [22]. One can further distinguish between the Service Providing Linking Interface (SPLIF) and the Service Requesting Linking Interface (SRLIF) of a real-time communication LIF. While the SPLIF offers the service of the component to all other components of the distributed system, the SRLIF requests services from other components.

Formally, we can formulate the component state $S_{\text{component}}$ of a component A at time t as following:

$$S^{A}_{\text{component}}(t) = \left(\bigcup_{i} S^{A^i}_{\text{interface}}(t) \right) \cup S^{A}_{\text{internal}}(t) \qquad (1)$$

where $S^{A^i}_{\text{interface}}(t)$ denotes the interface state of component A (either an SPLIF or an SRLIF) at time t with respect to interface $i, i \in \{1, 2, \ldots, n\}$ and $n \in \mathbb{N}$. The internal state $S^{A}_{\text{internal}}(t)$ of component A denotes the part of the state that is not made explicit through interfaces (e.g., temporary results, registers, program counter).

B. Interface State

Components are interconnected with other components by means of exchanging messages across LIFs [22] to realize emerging services. LIFs can be specified at two levels: the operational level incorporates syntactic and temporal aspects, while the meta-level denotes the semantics [25]:

1) **Syntactic Specification.** The syntactic specification defines the structure and name of the data elements exchanged via the interface. Thus, the concept of syntax

is used to construct structured information from basic information units.

2) **Temporal Specification.** The temporal specification determines the temporal sequence of message exchanges.
3) **Meta-Level Specification.** The semantic specification assigns a meaning to the structured information.

The specification of LIFs is a statement about the messages that arrive at SRLIFs and depart from SPLIFs. If the messages produced by a component comply to its specification at the two levels, a component is correct. Otherwise, the component exhibits a message failure and the component is denoted as faulty. Associated with such a message failure is an incorrect *interface state*, which is the state of a component as viewed from a particular interface [26]. In contrast to the internal state, the *interface state is accessible* to other components in the distributed system. As depicted in Figure 3 the interface state

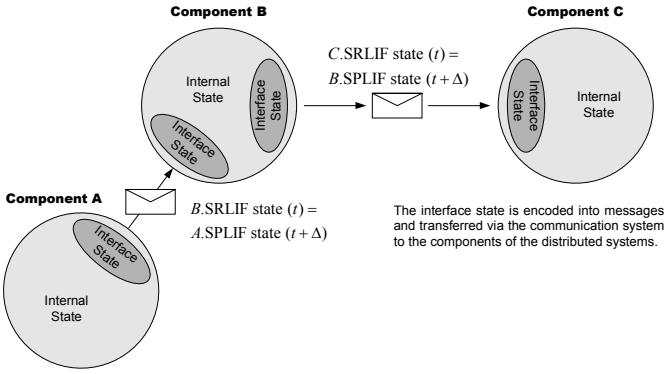

Fig. 3. Interface State

of each component is encoded into messages and transferred via the communication system. Thereby a replication of the interface state is performed. Note, that the internal component state remains hidden.

- **Interface State of SPLIFs.** While intermediate computational results (in general) only effect the internal state variables (i.e. the internal state) of a component, the final computational results are made available via the SPLIFs. Thus, the information from private, internal state variables is propagated into the component's interface state containing the set of state variables that are explicitly exported by a component allowing access by other components. In other words, the interface state of a component are those state variables that are transported to other components by exploiting the communication service.
- **Interface State of SRLIFs.** By exploiting the SRLIF, a component adapts its state variables with the state variables of the sending component. This way, a component acquires access to foreign state, which functions as input for subsequent computations – in the same way as the history of the component. By processing the interface state of an SRLIF the synchronized state of the sending component is propagated to the state of the receiving component and thus determines future behavior of the component provided via the SPLIF.

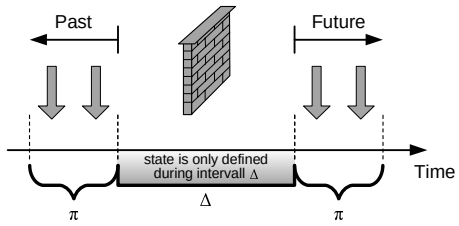

Fig. 4. Sparse Time Base

Due to the importance of the interface state concept, we also introduce the notion of *visible component state*, which consists solely of the interface state of a component (i.e. omitting the internal state). Formally, the visible component state $S^A_{\text{component,visible}}(t)$ of a component A at time t is defined as follows:

$$S^A_{\text{component,visible}}(t) = \left(\bigcup_i S^{A^i}_{\text{interface}}(t) \right) \tag{2}$$

C. Distributed State

If the time base in a distributed system is dense (the events are allowed to occur at any instant of the timeline), then it is in general not possible to generate a consistent temporal order on the basis of the time-stamps [23]. Due to the impossibility of synchronizing clocks perfectly and the denseness property of real time, there is always the possibility that a single event is time-stamped by two clocks with a difference of one tick. By introducing the concept of a *sparse time base* the ordering of events can be restored without execution of agreement protocols only based on timestamps [27]. In the sparse time model the continuum of time is partitioned into an infinite sequence of alternating durations of activity (π) and silence (Δ). Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice.

The activity intervals of the sparse time base form a synchronized system-wide action lattice. The interval of silence (Δ) on the sparse time base is a system wide consistent dividing line between the past and the future and the interval when the state of the distributed system can be defined. This consistent dividing line between the past and the future is illustrated in Figure 4.

At any point in time t, where t is an instant on the sparse time base corresponding to an interval of silence Δ, the union of the visible state of each component is denoted as the *distributed state*. During the interval of activity π the distributed state is undefined. The state of a distributed system $S_{\text{distributed}}(t)$ at time t can thus be devised as:

$$S_{\text{distributed}}(t) = \left(\bigcup_A S^A_{\text{component,visible}}(t) \right) \tag{3}$$

where $S^A_{\text{component,visible}}(t)$ denotes the visible state of a compo-

nent A at time t.

Adversely, the component state $S^{A}_{\text{component,visible}}(t)$ in a component A of the distributed system includes part of the distributed state $S_{\text{distributed}}(t)$. Through the exchange of information via the communication system, in each component a local manifestation of a part of the distributed state occurs. The part depends on the communication relationships between the components. This local manifestation of the distributed state will be exploited in Section IV for capturing certain types of diagnostic evidence, which function as an indicator for the occurrence of faults.

D. Consequences of Faults on the Distributed State

When a fault hits one or more constituting parts of the distributed system, a change of state can occur that leads to an unintended state denoted as an error [28]. Depending on the type of fault (e.g., internal or external fault, software or hardware fault), the unintended state will exhibit a characteristic manifestation in time, value and space. To capture the characteristics of the fault-induced distributed state changes, we introduce the concept of *fault pattern*. A fault pattern is the set of state variables that has been identified as subject to fault-induced state changes along with corresponding properties in value, space, and time. Different types of faults show different fault patterns on the distributed state.

- **Value Dimension.** The value dimension denotes a subset of the value domain of the selected state variables. This subset of the value domain is characteristic to the fault covered by the fault pattern.
- **Time Dimension.** The time dimension covers the persistence of the change in state. The time dimension makes statements about points in time, the durations, and the variability of the points in time and the variability of the durations of fault-induced state changes. Additional indicators on the time dimension are the time between successive fault-induced state changes and the frequency of state changes.
- **Space Dimension.** The space dimension specifies the locality and spatial expansion of the observable state changes. For example, the space dimension can encode whether a fault affects a single or multiple components and the physical proximity of the components hosting the changed state variables.

An important input for deriving fault patterns is the fault hypothesis. The fault hypothesis expresses the statements about the considered Fault Containment Regions (FCRs), the failure modes, the temporal properties, the failure frequency and the error detection latency [29]. In case the fault hypothesis states that a FCR with respect to software and hardware faults is a component, then a distinction between hardware and software faults is only possible by including the spatial dimension into the fault patterns.

Another example may be a fault hypothesis that states that the software has to be free of design faults (as required in some safety-critical systems). In this case only hardware faults need to be considered in case of component failure.

Dimension	Fault Patterns		
	Wearout	Massive Trasient	Connector Fault
Time	increasing frequency as time progresses	approximately at the same time (within a small delta)	arbitrary
Space	one component only	multiple components with spatial proximity	one component
Value	increasing deviation from correct value, at the verge of becoming incorrect	multiple bit flips	message omissions on a channel

Fig. 5. Summarized Fault Patterns

In the following we exemplify some typical fault patterns, which are summarized in Figure 5.

- **Wearout fault pattern:** A *wearout failure* is defined as a failure due to monotonic accumulation of incremental damage beyond the endurance of the material [30]. Such a wearout failure will exhibit a fault pattern typical for intermittent type faults [28]. This type of faults affects only a single component (space dimension) and reoccurs repeatedly at the same location at higher rates with decreasing intervals [31].
- **Massive transient disturbances fault pattern:** *Massive transient disturbances* (e.g., due to Electromagnetic Interference (EMI)) are an example for the class of faults typically affecting multiple components at the same time. EMI causes correlated effects on the entire system that usually cause no physical damage to hardware [32].
- **Connector Fault:** A *connector failure* may occur at an arbitrary point in time. Characteristic for the connector fault is the space and value dimension. A connector fault is assumed to affect only one component at a time and manifests itself as a message omission on a channel.

IV. Out-of-Norm Assertions

In the following section we will introduce Out-of-Norm Assertions (ONAs) as a generic mechanism operating on the consistent distributed state induced by a sparse time base for the encoding of fault patterns in the value, time and space domain. Besides supporting the detection of a violation of a component's service specification, ONAs establish means for the detection of system irregularities that cannot be forced into a bivalent assessment scheme at the time of occurrence. ONAs correlate spurious states in value, space and time in order to allow component assessment.

A. Definition of Out-of-Norm Assertions

The specification of a component describes the services that are offered via the LIFs to other components. Thus, it is possible to decide whether a component adheres to its specification or exhibits a behavior deviating from the intended service. In general the specification is a statement about services not about state. Since the provision of a correct service by a component implies a correct SPLIF interface state, the behavior of a component with respect to its service specification can only be checked by continuously evaluating the interface state. An interface state error is thus an indication of an upcoming failure of the respective service of a component.

In contrast to an interface state error, that is a definitive violation of the specification – since an interface state error corresponds to a component failure – an *anomaly* is an interface state that can only be judged as correct or incorrect at time of occurrence by the omniscient observer, but not by the computer system (e.g., due to missing a priori knowledge or redundancy to decide on the correctness). Only by continuously evaluating such anomalies over time, a (definite) classification of the experienced behavior, i.e. a judgement whether the intended service is provided, can be derived. Figure 6 shows this consideration. While incorrect interface states are perceived as failures of the respective component, anomalies are a deviation from the expected, regular interface states and require further assessment over time.

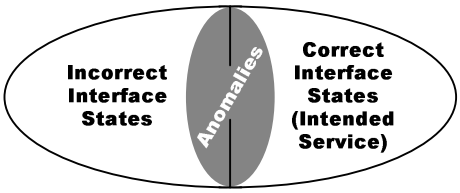

Fig. 6. State Space

We define an *Out-of-Norm Assertion (ONA)* as a predicate on the distributed system state that encodes a fault pattern in the value, time and space domain as introduced in Section III-D. ONAs are deterministically triggered, whenever all symptoms of a particular fault pattern are detected on the distributed state. A *symptom* is a set of interface state variables of a particular component that are monitored to detect deviations from the LIF specification. An ONA will likely be composed of more than one symptom, each operating on the interface state of different components. As depicted in Figure 7 ONAs can be hierarchically structured. This allows for the exploitation of identified symptoms for the implementation of different ONAs.

In contrast to conventional error detection techniques, ONAs do not provide a definite classification whether a component is correct or incorrect in case only a subset of the specified symptoms fire. In this case, we speak of an anomaly, i.e. we cannot ascribe the behavior of the component to a specific fault pattern. In order to decide on the correctness of a component, an assessment over time is necessary. The repeated evaluation of evidence gathered by ONAs provides the foundation for the analysis process that ultimately decides whether a component is correct. ONAs are gathering evidence in order to decide on a particular pattern affecting the state of the system. This process can be compared with a gathering evidence of different diagnostic techniques in medicine (e.g., temperature measurement, computer tomography, x-ray). In case sufficient evidence is gathered, a suspicion for a particular disease is confirmed or falsified.

Note, although an ONA fires deterministically, i.e. evaluates to either $\mathbb{T}$ or $\mathbb{F}$, the fault pattern encoded in the ONA may only be derived in a probabilistic manner (e.g., from field data). For

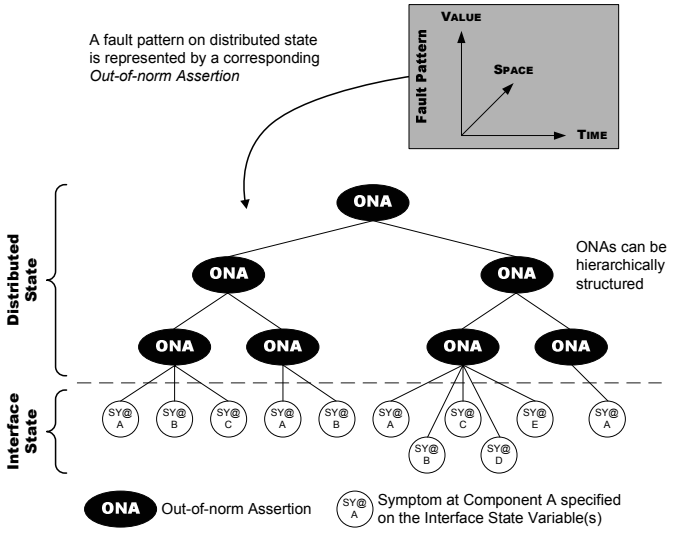

Fig. 7. Definition of Out-of-Norm Assertion

example, a connector failure can be detected by monitoring the communication channel in a distributed system. In case of replicated channels, a transient failure of one channel can be represented by a connector fault pattern. Since, connector failures account for a significant portion of communication faults [33] it follows, that a message omission failure on one channel is likely due to a connector fault (e.g., loose contacts). However, there will be message omission failures resulting also from other faults, although with a significantly lower probability.

By interpreting the information of all components of the system, correlated failures can be identified that allow to distinguish between external and internal faults. For example, while transient external faults (e.g., due to EMI) randomly effect components, intermittent type faults occur repeatedly at the same component at a higher rate [31] (e.g., solder joint cracks). This distinction is especially useful in case of integrated architectures such as Integrated Modular Avionics (IMA) [34], where components are shared among multiple applications. Here, the determination of experienced failures to a particular application or to a set of applications is important to decide whether a software or hardware fault is active.

The evaluation process performed by the diagnostic subsystem is illustrated in Figure 8. The evaluation process is based on a consistent notion of state, which is provided through the action lattice of the sparse time base. The arrows in Figure 8 indicate the assessment trajectories. At first both arrows show component conformance with the specification, i.e. correct interface states. As time progresses arrow A exhibits an increasing confidence for a violation of the specification, while arrow B indicates a component behavior in accordance with the specified service.

B. Classification of Symptoms: Self- vs. Cross-Checking

If a component evaluates its own symptoms, we denote this process as *self-checking*. However, when the component is

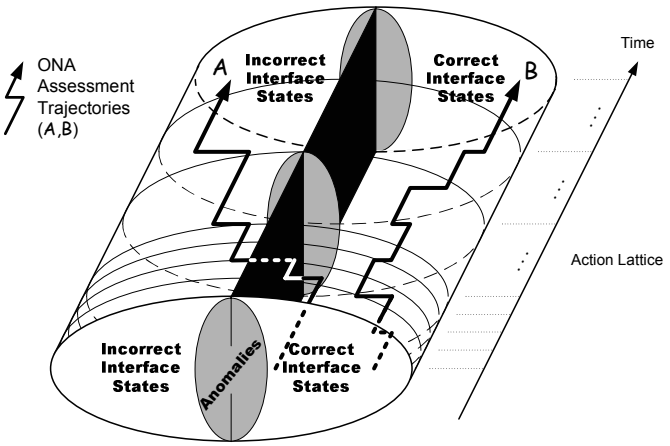

Fig. 8. Assessment Process

affected by a fault, it cannot be assumed that the error detection mechanisms within this component are unaffected by this fault. Thus, there is always the possibility that the judgement of a faulty component on its correctness is misleading.

In contrast, the interface state is revealed to other components via the exchange of messages through the communication system and can be checked independently by all other nodes. We denote this type of checking of symptoms with respect to specification conformity as *cross-checking*. The validity of such symptoms that can be tested by other components is more trustworthy than the results of checks performed by the node under inspection. This concept conforms to the established architectural principle in safety-critical architectures of mutual error detection [35].

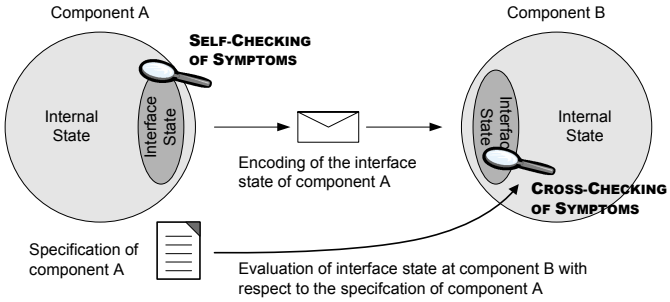

Fig. 9. Self-Checking vs. Cross-Checking

As stated before, the interface state is transferred to all receiving components via the communication service (i.e. the interface state is encoded into messages), it follows that detection techniques on the interface state at the receiving component are a mechanism to assess the correctness of the sending component. As illustrated in Figure 9, component B is able to apply error detection on the interface state with respect to the specification of component A.

Consequently, by applying ONAs we introduce symptoms as *constraints on the interface state* to assess the condition of a system component. Fundamental to this concept is the fact that the interface state of the sending node (Component A) is mapped onto the interface state of the receiving node (Component B) as indicated in Figure 10. As depicted, a fault in component A causes an error in the interface state of the LIF. Subsequently, this error causes a message failure (e.g., timing failure, value failure). Consequently, a failure of the sender manifests itself as an error in the interface state of the receiver (with respect to the specification of the sending node). In case the interface error with respect to the

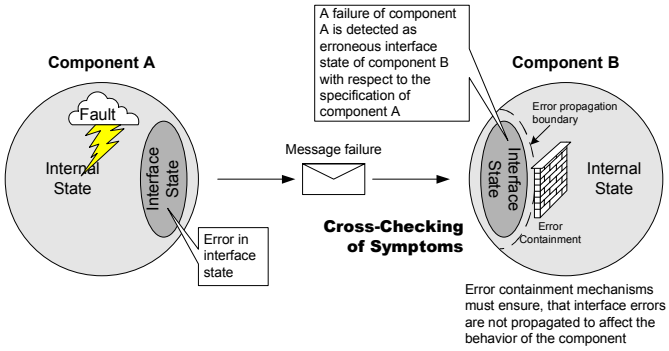

Fig. 10. A Failure of the Sender is Detected as an Error in the Interface State of the Receiver

specification of component A remains undetected, the error propagates to the state of component B and can lead to a consequent failure of component B. For that reason error containment mechanisms must ensure that interface errors cannot propagate to affect the service of the component [36].

V. Application of Out-of-Norm Assertions for Solving Prevalent Diagnostic Problems

ONAs can help to solve existing diagnostic problems of distributed embedded real-time systems. Among the most important problems are:

A. Reduction of the Trouble Not Identified (TNI) phenomenon

Currently, industry has significant problems detecting and identifying electronic devices that cause transient system failures in electronic systems. This so called *TNI phenomenon* is an increasing problem in automotive and avionic electronics with major economic implications [3]. This lack of information often results in unnecessary replacements of working components [4], [37]. Even worse, in many cases the faulty component causing the failure remains in the systems. As a consequence, these spurious failures have a lasting effect on the customers trust in the product [5] and the reputation of the manufacturer. In the avionic domain the No Fault Found (NFF) problem is estimated to account for approximately 300 million dollar per year (equivalent to six new A320 aircrafts). With an average cost of 800$ per removal of a single Line Replaceable Unit (LRU), the is a huge potential of cost reduction [38]. The economic impact in the automotive industry is even more dramatic.

The task of system diagnosis is to assess the operational state of a system. In case of a distributed system, diagnosis must operate on the distributed state to diagnose correlated

errors and to indisputably judge about the functional correctness of the constituting parts of the systems, i.e. the components. In case the diagnosis subsystem operates on the local states, only those errors can be traced, that can be classified without knowledge about other components. For instance, in the automotive domain the OBD systems tend to analyze local information in contrast to global information to assess the health condition of each ECU. In combination with a system architecture that provides only limited control of error propagation, such a diagnostic subsystem rather provides hints about possible faults than an exact identification of the component that needs to be replaced.

Consider for example transient internal versus transient external faults. Both types of faults are frequently causing spurious component failures in distributed embedded real-time systems. While an ONA covering transient internal faults restricts the space domain to only one Field Replaceable Unit (FRU) with respect to hardware faults, an ONA for the detection of transient external faults needs to cover multiple FRUs. This way the spatial proximity is taken into account, as a prerequisite for any analysis process. Since the symptoms of an ONA can be verified by multiple components by cross-checking the respective interface state, misleading error messages of faulty components can be identified and precluded from any further assessment.

B. Condition-Based Maintenance

Time-Based Maintenance (TBM) is increasingly being replaced by Condition-Based Maintenance (CBM), to reduce costs and to improve reliability and system performance [39]. Originally introduced to the avionics domain, this new paradigm is more and more accepted in the automotive industry. Besides the reduction of cost of ownership (service only what is needed) the possibility of collecting accurate field data (i.e. *engineering feedback*) is one of the major benefits from this maintenance approach. In addition, the customer trust in the car [5] will be increased, since component replacements can be performed by the service technicians before the owner of the car is informed by the car's OBD system.

In order to adopt CBM for electronic systems suitable indicators for degradation or wearout must be identified and analyzed to detect deviations from sound operation. For example in machinery vibration, thermal, and lubricant analysis are good indicators for possible defective conditions [40]. One of these indicators in the electronic domain is the increasing rate of transient failures [31].

ONAs can be deployed to detect deviations of components that indicate future component failure and transient errors resulting from intermittent type faults. The symptoms of ONAs are suited to detect those system states that are correct but at the verve of become incorrect in the near future. The continuous analysis of the acquired information by the ONAs allows for preventive measures at an early stage.

A wearout fault affects only a single FRU with respect to hardware faults and reoccurs repeatedly at the same location at higher rates with decreasing intervals [31]. Consequently, the time domain is of great significance in the identification of wearout phenomenons in order to realize CBM, whereas the spatial dimension of is usually restricted to one FRU (see also the wearout fault pattern in Figure 5).

Consider for example clock synchronization. In case the correction term of the clock synchronization algorithm is larger than $\Pi/2$, where Π denotes the precision, the node raises a clock synchronization error and is excluded from the set of operational components. ONAs allow to detect such a drift (e.g., due to wear-out effects or rapid temperature change) at an early stage. In case the correction term is at the upper or lower limit of the correct range $[-\Pi/2, \Pi/2]$ a value anomaly with respect to the quartz has been identified. The evaluation process to identify wearout phenomenons is performed by the diagnostic subsystem and is illustrated in Figure 11. The

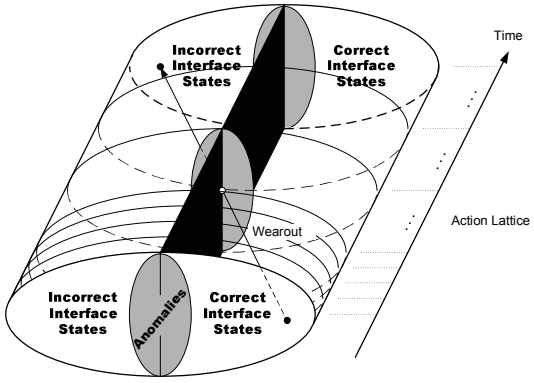

Fig. 11. Assessment of a Wearout Phenomenon

arrow in Figure 11 indicates this wearout phenomenon, i.e. the increasing drift rate of the quartz. At first the rate of the quartz is in conformance with its specification, but as time progresses, the quartz frequency drifts away from the specified frequency. This can be made obvious by measuring the clock correction term in case the system supports a global time base through clock synchronization. As soon as the correction term value is at the borders of the synchronization interval, an ONA is raised (i.e. this is a correct state). If the ONA fires repeatedly within an a priori defined interval, the diagnostic subsystem can conclude that the component will fail in the near future and indicate the maintenance engineer to change the component in time.

C. Application for Integrated Architectures

Integrated systems such as IMA [34] in avionics are characterized by the integration of multiple application subsystems within a single distributed computer system. Integrated systems promise massive cost savings through the reduction of resource duplication. In addition, integrated systems permit an optimal interplay of application functions, reliability improvements with respect to wiring and connectors, and overcome limitations for spare components and redundancy management. Integrated architecture provide a finer granularity of diagnostic information than federated systems. The decomposition of the overall system into application subsystems

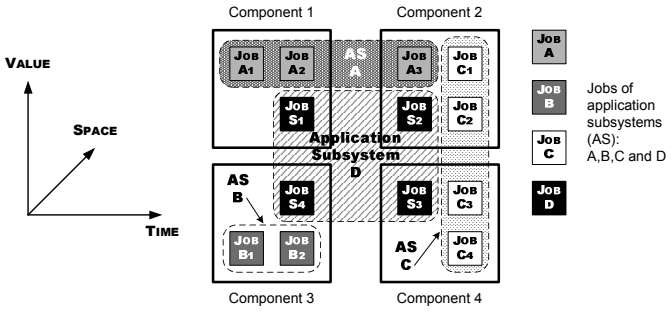

Fig. 12. The Importance of the Space Domain in Integrated Architectures

with respective jobs (i.e. units of execution) is a key element for a more precise differentiation of experienced faults. By including the three dimensions of time, value, and space into the judgment process, a discrimination into internal hardware faults, external hardware faults and software faults is possible. For instance, consider the system depicted in Figure 12. In case a software fault hits the jobs A_1, A_2, and A_3 of the application subsystem A, the fault affects only the application subsystem A, since the error containment mechanism ensure that this fault cannot propagate to other application subsystem. In contrast, in case a hardware fault hits a component hosting multiple jobs of different application subsystem, it is very likely that the impact of this fault is not limited by application subsystem borders. A hardware fault will cause multiple jobs hosted on one component to fail (e.g., the jobs A_3, C_1, C_2 and S_2 on component 2 in Figure 12).

In analogy, other fault classes can be covered via Out-of-Norm Assertions (ONAs) that are continuously evaluated against the distributed state of the system.

Since Out-of-Norm Assertions operate primarily on the interface state (as the component's representation of the distributed state), no details about the internals of a component are necessary to apply this proposed mechanism. The interface specification, i.e. the service specification of the component, is sufficient to design and deploy ONAs. This allows for an effective detection mechanism for the diagnostic subsystem of an integrated architecture and maintain Intellectual Property (IP) protection at the same time.

VI. Conclusion

In order to solve prevalent diagnostic problems in modern electronic systems, emphasis must be set on the identification of faults leading to failures and anormal behavior. This goal can only be achieved by the deployment of diagnostic techniques operating on the consistent distributed state resulting from a sparse time base. This, on contrast to diagnostic systems revealing only internal component state, allows to distinguish between FRU failures originating from external faults (e.g., EMI) from those failures that can be extinguished by changing the respective FRU.

The introduced concept of Out-of-Norm Assertions (ONAs) as an detection mechanisms operating on the distributed state encodes identified fault patterns in the time, value and space domain. In contrast to conventional assertions, that typically check only the value dimension, ONAs can be applied to a variety of prevalent diagnostic problems industry is currently facing. By solely operating on the interface state the symptoms of ONAs can be independently checked by the components of the distributed systems. This way ONAs allow to refrain from traditional OBD systems that, in general, rely on information gathered by monitoring the internal component states. ONAs can in particular be applied in integrated architectures as part of the diagnostic infrastructure. Here, a precise differentiation in the space domain plays a central role in diagnosing faults affecting the system. This way ONAs can be deployed to realize *the detection of any single anomaly* design principle for safety-critical distributed systems.

Acknowledgments

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

References

[1] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.

[2] The Hansen Report on Automotive Electronics, November 2002. Portsmouth NH USA, `www.hansenreport.com`.

[3] D.A. Thomas, K. Ayers, and M. Pecht. The 'trouble not identified' phenomenon in automotive electronics. *Microelectronics Reliability*, 42:641–651, 2002.

[4] M. Mateos, P. Robin, S. Sauvage, V. Joloboff, G. Madhusudan, and Y. Bennani. Environment for evolutionary automotive diagnosis. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[5] I. Berger. Can you trust your car? *IEEE Spectrum*, 39(4):40–45, April 2002.

[6] G.T. Jermy, R.C. Castle, C.F. Gimblett, and R. Chakrabarti. Monitoring out of normal conditions in repetitive cycle production machinery. In *Proceedings of the Fifth International Conference on Factory 2000*, pages 29–33. IEEE, April 1997.

[7] T. Hoare. Assertions. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proceeding of the Second International Conference on Integrated Formal Methods*, LNCS 1945, pages 1–2, Dagstuhl Castle, Germany, January 2000. Springer-Verlag Heidelberg.

[8] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceeding of the Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] K.R. Apt. Ten years of hoare's logic: A surveypart i. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[11] J.M. Adams, J. Armstrong, and M. Smartt. Assertional checking and symbolic execution: An effective combination for debugging. In *Proceedings of the 1979 annual conference*, pages 152–156. ACM Press, 1979.

[12] B.M. McMillin and L.M. Ni. Executable assertion development for the distributed parallel environment. In *Proceedings of the Twelfth International Computer Software and Applications Conference (COMPSAC 88)*, pages 284–291. IEEE, October 1988.

[13] D.S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th international conference on Software engineering*, pages 92–104. ACM Press, 1992.

[14] T. Brotherton and T. Johnson. Anomaly detection for advanced military aircraft using neural networks. In *Proceedings of the IEEE Aerospace Conference*, volume 6, pages 3113–3123. IEEE, March 2001.

[15] R.A. Maxion and K.M.C. Tan. Anomaly detection in embedded systems. *IEEE Transactions on Computers*, 51(2):108–120, February 2002.

[16] L.C. Jaw and D.N. Wu. Anomaly detection and reasoning with embedded physical model. In *Proceedings of the IEEE Aerospace Conference*, volume 6, pages 6–3073–6–3081. IEEE, March 2002.

[17] R.R. Lutz and I.C. Mikulski. Evolution of safety-critical requirements post-launch. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 222–227. IEEE, August 2001.

[18] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, pages 24–33. IEEE, June 2000.

[19] W. Waldeck. Diagnostic protocol challenges in a global environment. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[20] International Organization for Standardization. Keyword Protocol 2000, ISO 14230. ISO, `www.iso.ch`, 1999.

[21] M. D. Mesarovic and Y. Takahara. *Abstract Systems Theory*, chapter 3. Springer-Verlag, 1989.

[22] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[23] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[24] A. Ran and J. Xu. Architecting software with interface objects. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pages 30–37, Herzliya, Israel, June 1997. Software Technol. Lab., Nokia Res. Center, Espoo.

[25] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[26] M.-C. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, December 2002. Available as Research Report 54/2002 at http://www.vmars.tuwien.ac.at.

[27] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of 12th International Conference on Distributed Computing Systems*, Japan, June 1992.

[28] J.C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer Verlag, Vienna, Austria, 1992.

[29] H. Kopetz. The fault hypothesis for the time-triggered architecture. In *Proceedings of the IFIP World Computer Congress*, 2004.

[30] A. Ramakrishnan et. al. *The Avionics Handbook*, chapter Electronic Hardware Reliability. CRC Press LCC, 2001.

[31] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 205–209. IEEE, 2002.

[32] H. Kim, A.L. White, and K.G. Shin. Effects of electromagnetic interference on controller-computer upsets and system stability. *IEEE Transactions on Control Systems Technology*, 8(2):351–357, March 2000.

[33] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In *Proceedings of the 45th IEEE Holm Conference on Electrical Contacts*, pages 146–152, Pittsburgh, PA, USA, October 1999. Dept. of Mech. Eng., Southampton Univ.

[34] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.

[35] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems*, April 2003.

[36] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. In *Proceedings of the IEEE*, volume 82 of *1*, pages 25–40, January 1994.

[37] R. Tappe and D. Ehrhardt. Dynamic tests in complex systems. In *Proceedings of the International Test Conference*, pages 609–614. IEEE, 2001.

[38] M. Lorell, J. Lowell, M. Kennedy, and H.P. Levaux. *Cheaper, Faster, Better? Commercial Approaches to Weapons Acquisition*. RAND Corporation, 2000.

[39] C. Teal and D. Sorensen. Condition based maintenance [aircraft wiring]. In *Proceedings of the 20th Conference on Digital Avionics Systems, DASC*, volume 1, pages 3B2/1–3B2/7, October 2001.

[40] A.G. Starr. A structured approach to the selection of condition based maintenance. In *Proceedings of the Fifth International Conference on Factory 2000 - The Technology Exploitation Process*, pages 131–138, 1997.