Testing Safety Properties of Cyber-Physical Systems with Non-Intrusive Fault Injection – An Industrial Case Study

Joachim Fröhlich[1(✉)], Jelena Frtunikj[2], Stefan Rothbauer[1], and Christoph Stückjürgen[1]

[1] Siemens AG, Otto-Hahn-Ring 6, 81739 Munich, Germany
froehlich.joachim@siemens.com
[2] Fortiss GmbH, Guerickestrasse 25, 80805 Munich, Germany

Abstract. Non-intrusive, deterministic fault-injection tests provide evidence for making reliable statements about the behavior of safety-critical, real-time systems in the presence of software faults and component failures. These tests are derived from system safety requirements for the detection and handling of value and time errors. That the approach presented here works for distributed, time-triggered systems that process data cyclically and reserve resources exclusively for testing purposes has been demonstrated by an industry study confirming the feasibility of the concepts for a fail-operational electric car.

Keywords: Cyber-physical system · Fault-tolerant system · Safety requirement · Fault injection test

1 Introduction

An open question for fault-tolerant, cyber-physical systems is how to reliably demonstrate their safety properties. Since the root causes of failures are faults, fault injection is an established practice for testing systems. There are promising approaches for injecting hardware and software models with faults without adversely affecting the simulation time during simulation runs [1,9]. However simulation-based fault-injection of executable system models ultimately fails to hold for operation systems, essentially for two reasons: (1) By their very nature, system models abstract implementation details and cannot be fully accurate in every aspect for operational systems in the field that need to execute under tight real-time constraints. (2) System environment models are hard to parametrize accurately with realistic simulation data. Software Implemented Fault Injection (SWIFI) is an established technique for fault injection into operational software systems, but has a significant disadvantage: SWIFI changes the timing behavior due to probe effects. The same disadvantage applies for tests in general and in particular for fault injection tests that stimulate and check the behavior of distributed systems online solely on the network level (see for example [6]). Another issue is the limited observability and controlability of systems under test. The limits of SWIFI tests define the fundamental objective of this work: to accurately test real-time systems with tight schedules while running free of side effects. The scope of the system which these tests observe and control shall be maximized to let concise tests illuminate otherwise obscure locations and behaviors of the system under test.

The contributions of this work are twofold: (1) demonstration of deterministic tests of safety requirements to provide reliable statements on fault-tolerant systems in operation; (2) an explanation of when, where, and why such tests provide reliable statements.

This paper is structured as follows: Sect. 2 introduces a consistent set of terms for characterizing target systems, assigned safety requirements and tests. Section 3 presents safety requirements used as driving examples. Section 4 characterizes target systems that implement fault-tolerance mechanisms and enable the execution of fault-injection tests without inadmissible probe effects. Section 5 specifies tests in ALFHA[1] [5] that verify the fulfillment of the safety requirements. The tests are executed with VITE[2]. Section 6 checks the plausibility of results that the test system produces. We use RACE[3] as our reference target system [2,4,8].

2 Terms

Throughout the paper we use a consistent set of terms for characterizing target systems, assigned safety requirements and tests for these requirements. Some terms are implemented as system predicates and used in test procedures (Sect. 5), such as **Platform Node**, **Dual Platform Node** and **Master Host**. Figure 1 provides an overview of selected terms explained in the following.

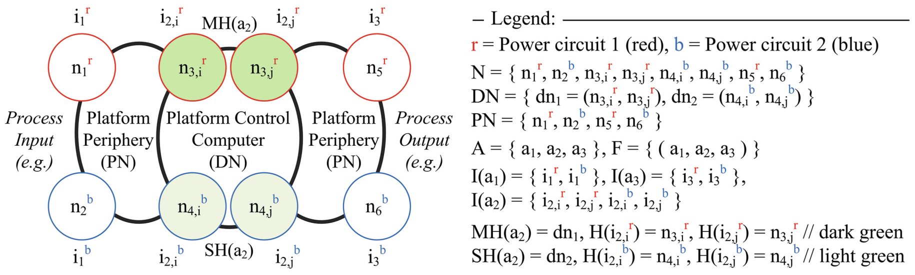

i_1^r
$i_{2,i}^r$
MH(a_2)
$i_{2,j}^r$
i_3^r
n_1^r
$n_{3,i}^r$
$n_{3,j}^r$
n_5^r
Process Input (e.g.)
Platform Periphery (PN)
Platform Control Computer (DN)
Platform Periphery (PN)
Process Output (e.g.)
n_2^b
$n_{4,i}^b$
$n_{4,j}^b$
n_6^b
i_1^b
$i_{2,i}^b$
SH(a_2)
$i_{2,j}^b$
i_3^b
– Legend:
r = Power circuit 1 (red), b = Power circuit 2 (blue)
$N = \{ n_1^r, n_2^b, n_{3,i}^r, n_{3,j}^r, n_{4,i}^b, n_{4,j}^b, n_5^r, n_6^b \}$
$DN = \{ dn_1 = (n_{3,i}^r, n_{3,j}^r), dn_2 = (n_{4,i}^b, n_{4,j}^b) \}$
$PN = \{ n_1^r, n_2^b, n_5^r, n_6^b \}$
$A = \{ a_1, a_2, a_3 \}, F = \{ (a_1, a_2, a_3) \}$
$I(a_1) = \{ i_1^r, i_1^b \}, I(a_3) = \{ i_3^r, i_3^b \},$
$I(a_2) = \{ i_{2,i}^r, i_{2,j}^r, i_{2,i}^b, i_{2,j}^b \}$
$MH(a_2) = dn_1, H(i_{2,i}^r) = n_{3,i}^r, H(i_{2,j}^r) = n_{3,j}^r$ // dark green
$SH(a_2) = dn_2, H(i_{2,i}^b) = n_{4,i}^b, H(i_{2,j}^b) = n_{4,j}^b$ // light green

Fig. 1. An example target system with platform network (rings) (Color figure online)

[1] Assertion Language for Fault-Hypothesis Arguments.
[2] Verification and Integration Testing Environment, www.aviotech.de.
[3] Reliable Automation and Control Environment, www.projekt-race.de/en.

Platform Mechanism. Code block that implements a fault-tolerance mechanism such as, in general terms, error detection (ed), error recovery (er) and/or error mitigation (em), or a basic operating system mechanism such as input/output data processing and operation scheduling. Let $M = \{ed, er, em, ...\}$ denote the set of all platform mechanisms. Each platform mechanism $m \in M$ can be instantiated to an instance $i \in I(m)$, with $I(m)$ denoting the set of all instances $\{i_1, i_2, ...\}$ of m.

Platform Node. Node for short. Computer n consisting of a CPU, a clock generating cyclic ticks of constant duration, main memory, an access point to the network of all platform nodes $N = \{n_1, n_2, ..., n_n\}$ of the target system, an access point to a test network (not shown in Fig. 1 but in Fig. 3), and deployed instances of platform mechanisms. Each computer n is connected to either the "red" power circuit (n^r) or to the "blue" power circuit (n^b).

Platform. Instantiated and deployed platform mechanisms providing together fault-tolerance (safety) mechanisms to platform applications.

Platform Application. Application for short. Code block that uses platform mechanisms. Let $A = \{a_1, a_2, ...\}$ denote the set of all platform applications. Each platform application $a \in A$ can be instantiated to an instance $i \in I(a)$ that is deployed on a platform node. $I(a)$ denotes the set of all instances $\{i_1, i_2, ...\}$ of a. For clarity, we sometimes write i^r or i^b if the underlying platform node n is connected to the "red" power circuit (n^r) or to the "blue" power circuit (n^b).

Dual Platform Node. Dual node for short. Pair of platform nodes $dn = \{n_i, n_j\}$, with $n_i, n_j \in N$ running in lockstep mode and being connected to the same power circuit. In context of a dual platform node n_i and n_j are also called twin nodes.

Platform Control Computer. All dual platform nodes $DN = \{dn_1, dn_2, ..., dn_n\}$, with $\forall dn_i, dn_j \in DN : dn_i \cap dn_j = \emptyset$; that is, dual platform nodes run in disjoint node pairs.

Platform Periphery. All periphery nodes $PN = \{n : n \in N \wedge \nexists dn \in DN : n \in dn\}$. In other words, a node n belongs either to the platform periphery (sensing process inputs or controlling process outputs) or to the platform control computer.

Node Cycle. Instant of a cyclic process running in linear time [7] on a platform node, also called local instant. The clock of platform node n generates subsequent cycle numbers defined as numerical time series of node cycles $clock(n) = (0, 1, 2, ...)$. Cycle numbers of two different platform nodes n_i, n_j can differ at the same global instant, e.g., when the platform is up and running and when n_i started before n_j or when the clocks of n_i and n_j drift.

Platform Cycle. Instant of a cyclic process running in linear time on the platform, also called global instant. At platform start, the cycles generated by the clock of the first started platform node n determines the numerical time series of platform cycles $clock = (0, 1, 2, ...)$.

Variable. Addressable location $v(n)$ in the data segment of the main memory of platform node n. Platform mechanisms and platform applications exchange values via variables during a cycle and between cycles. We denote the value of a variable $v(n)$ at cycle x as $v(n)_x$, or simply v_x if n is irrelevant. Depending on the context, x denotes a (local) node cycle or a (global) platform cycle. Variables also take on values of input signals (process inputs originating from sensors in the platform periphery) and output signals (process outputs targeting actuators in the platform periphery).

Data Store. Section $V(n)$ in the data segment of the main memory of node n which contains all variables; that is, $V(n) = \{v_1(n), v_2(n), ...\}$. The data store of the platform contains all variables in the data stores of all nodes; that is, $V = \bigcup_{n \in N} V(n)$. Typically only a subset $W = \{v_1, v_2, ...\}$ of all variables V is in the test scope, that is $W \subseteq V$ and v_1, v_2 are different variables of possibly different nodes. The data stores of all nodes contain some common variables with node-specific values, e.g., node cycle and node state.

Trace. Chronologically ordered values of all variables in the test scope denoted as $(W_i)_{i=x..y} = (W_x, W_{x+1}, ..., W_y)$. Depending on the context, x and y denote (local) node cycles or (global) platform cycles.

System Function. All functionally coherent platform applications which together transform input signals from system sensors (related to nodes in the platform periphery) to output signals for system actuators (related to nodes in the platform periphery) defined as $F \subseteq A \times A \times ... \times A$. Instances of one or more platform applications instantiate a concrete system function.

Host. Platform node n that executes an instance i of platform application a at cycle x; that is, $H(i)_x = n \in N$. If x is not relevant then we write $H(i)$.

Master Host. Dual platform node MH whose nodes n_i and n_j both execute instances i_i and i_j of the same platform application a at cycle x; that is, $MH(a)_x = (n_i, n_j) \in DN$ with $i_i, i_j \in I(a)$, so that $H(i_i)_x = n_i$ and $H(i_j)_x = n_j$. $MH(a)$ operates as an open gate in the sense that $MH(a)$ transports signals or data to (input) and from (output) application a.

Slave Host. Dual platform node SH whose nodes n_i and n_j both execute instances i_i and i_j of the same platform application a at cycle x in hot-standby mode to $MH(a)_x$; that is, $SH(a)_x = (n_i, n_j) \in DN$ with $i_i, i_j \in I(a)$, so that $H(i_i)_x = n_i$ and $H(i_j)_x = n_j$. In contrast to $MH(a)$, $SH(a)$ operates as a half-side open gate in the sense that $SH(a)$ only transports signals or data to (input) application a.

3 Safety Requirements

3.1 Application Context

Our example system is an electric car built using software-intensive electronic devices. Safety-critical car functions, such as steering and braking, must be highly available and work reliably. In the following, we consider only car steering. In basic configuration, the steering system takes input from the driver, i.e., steering wheel position, and translates it into control commands for car wheels. An advanced variant of the steering system is controlled by additional parameters such as car speed, weight, weight distribution, yaw angle, and road and weather conditions. Regardless of the variant, the steering function, as well as the communication network and the on-board power supply, must stay operational in the presence of permanent or temporary faults in steering and non-steering car components.

3.2 System Scope

We assume that every fault-tolerant function roughly consists of three parts: input from the platform periphery (Fig. 1, sensors on the left), data processing in the platform control computer (Fig. 1, central nodes) and output to the platform periphery (Fig. 1, actuators on the right). Hence, the steering function consists of three platform applications $F = \{(sws, sc, wc)\}$ with: (1) *steering wheel sensing* (sws) having two redundant instances $I(sws) = \{sws^r, sws^b\}$ on two redundant steering wheel sensors $n_1^r, n_2^b \in PN : n_1^r = H(sws^r) \wedge n_2^b = H(sws^b)$; (2) central *steering control* (sc) having four redundant instances $I(sc) = \{sc_i^r, sc_j^r, sc_i^b, sc_j^b\}$ on four pairwise redundant, central nodes $(n_{3,i}^r, n_{3,j}^r), (n_{4,i}^b, n_{4,j}^b) \in DN : n_{3,i}^r = H(sc_i^r) \wedge n_{3,j}^r = H(sc_j^r) \wedge n4_i^b = H(sc_i^b) \wedge n4_j^b = H(sc_j^b)$ and (3) *wheel controlling* (wc) having two redundant instances $I(wc) = \{wc^r, wc^b\}$ on two redundant steering boxes $n_5^r, n_6^b \in PN : n_5^r = H(wc^r) \wedge n_6^b = H(wc^b)$. Redundant communication links and redundant power circuits complete the system. The following requirements concern availability properties of safety-critical system functions and thereby the steering function.

3.3 Requirement R1: Redundant Input Signals

R1.1 Safety property: Continuous data available. Host $n = H(i)_x$ shall provide a signal value to instance i of data processing application a in each node cycle. The signal value shall be free from those errors that platform mechanisms are responsible to detect and process.

For example, for car steering we assume the difference between two succeeding steering angles (v_x, v_{x-1}) of a safe longitudinal movement to lie within variable limits, that is, $|v_x - v_{x-1}| \leq delta$, even when a redundant steering wheel sensor fails. Driving situations, physical values and technical properties determine *delta*.

R1.2 Error detection. Host $n = H(i)_x$ shall check signal value $v(n)_x$ for errors that the platform mechanisms shall detect before providing $v(n)_x$ to i.

R1.3 Error mitigation. Host $n = H(i)_x$ shall provide signal value $v_2(n)_x$ to i when $v_1(n)_x$ is missing (no signal value received in cycle x) and if $v_2(n)$ is redundant to $v_1(n)$ and free of errors. Redundant signals $v_1(n)_x, v_2(n)_x$ from redundant senders $n_1, n_2 \in PN$ reach n in redundant, local variables $v_1(n), v_2(n)$.

R1.4 Sender abstraction. Instance i of application a cannot distinguish redundant signal values $v_1(n)_x$ and $v_2(n)_x$.

3.4 Requirement R2: Fail-Operational Data Processing

R2.1 Safety property: Master host available. Exactly one master host $MH(a)_x$ shall execute application a at platform cycle x.

R2.2 Error detection. Slave host $SH(a)$ shall detect the failed $MH(a)$ within $d > 0$ cycles, that is, in the interval from cycle $x+1$ to cycle $x+d$, when $MH(a)$ fails at platform cycle x.

R2.3 Error recovery. Slave host $SH(a)$ shall become $MH(a)$ within $s > 0$ cycles, that is, in the interval from cycle $x + d' + 1$ to cycle $x + d' + s$, when $SH(a)$ has detected the failed $MH(a)$ after d' cycles, with $d' \leq d$, and if the master-selection strategy selects $SH(a)$.

R2.4 Safety property: Master unavailable. Application a can run without $MH(a)$ for $d + s$ number of cycles, that is, in interval from cycle x to cycle $x + d + s$. Properties of the containing system function F and the situation dependent system environment determine the durations d and s.

R2.5 Host abstraction. Instances of application a cannot distinguish $MH(a)$ from $SH(a)$ at platform cycle x.

4 Target System

4.1 Platform Safety Mechanisms

The car system must stay fail-operational. To operate dependably such systems are realized as distributed, redundant components with replicated communication channels and redundancy control to tolerate all faults. System functions rely on redundancy handling and fault processing mechanisms built into the system platform. These mechanisms factored out into the platform simplify the implementation, integration, and testing of platform applications.

The heart of the example target system is the platform control computer built of several dual nodes (DN in Fig. 1). Single or redundant sensors and actuators in the platform periphery (PN in Fig. 1) connect the system to the system environment. Platform safety mechanisms ($\{ed$ = error detection, eh = error handling, ...$\}$) are instantiated once for each node. They automatically detect and handle value errors and time errors [3] or combinations thereof (Table 1).

Safe steering, for example, relies on the availability, reliability and integrity of the underlying system platform. In case of an inconsistency in the platform control computer, the faulty node immediately backs out so as not to jeopardize steering. For detecting inconsistencies, dual nodes pairwise monitor input data, output data and node states in every cycle (Table 1: f). If the inconsistent dual node is the master host of the central *steering control*, cyclic exchange of platform states and checks within all other dual nodes detect the faulty master host (Table 1: g). Then one of the hot standby slave hosts, still exchanging platform states, takes over the role of the master host (Table 1: j, k). As the steering function must constantly work alongside the redundant steering-wheel sensor,

Table 1. Detecting (ed) and handling (eh) of value (V) and time (T) errors

Platform mechanisms		Examples	V	T
Data plausibility	ed	a. Host checks value range in cycle	+	-
		b. Host checks value delta in subsequent cycles	+	-
Protocol integrity	ed	c. Host checks CRC of frames in cycle	+	+
		d. Host checks frame counters in subsequent cycles	+	+
		e. Host checks frame arrival time	-	+
Node integrity	ed	f. Nodes of dual nodes cyclically compare status	+	+
Platform integrity	ed	g. Dual nodes cyclically compare status	+	+
Vote signals (error mitigation)	eh	h. Host selects one of several redundant signals	+	+
Compensate signal (error mitigation)	eh	i. Host provides safe signal: last valid or default	+	+
Reconfigure platform (error recovery)	eh	j. Dual nodes determine one master host	+	+
		k. Dual nodes isolate faulty nodes	+	+

the platform mechanisms of the platform control computer check steering angles (and signal values in general, Table 1: *a*–*e*), vote, and select one per cycle to ensure that instances of the central *steering control* application obtain quality signal values in every cycle (Table 1: *h*).

4.2 Non-Intrusive Test Probe Mechanism

For demonstrating system safety in different system configurations of varying degrees of redundancy, the system platform must enable by design the test system to non-intrusively monitor and manipulate signal values, communication packets, system states and data quality indicators. Tests must be able to intervene simultaneously and instantaneously in different nodes. A target system is testable if it permits these interventions without accidentally altering system functionality and timing—neither in lab tests nor in field tests. The following properties of the system platform meet these requirements (Fig. 2):

Time-triggered architecture. Time-triggered systems behave deterministically because systems control events and not vice versa (as in event-triggered systems). Hence schedulers activate instances of platform applications and platform mechanisms in a time-triggered way.

Node data store. Instances of platform applications and platform mechanisms, on each node, communicate via a data store. A node data store captures signal

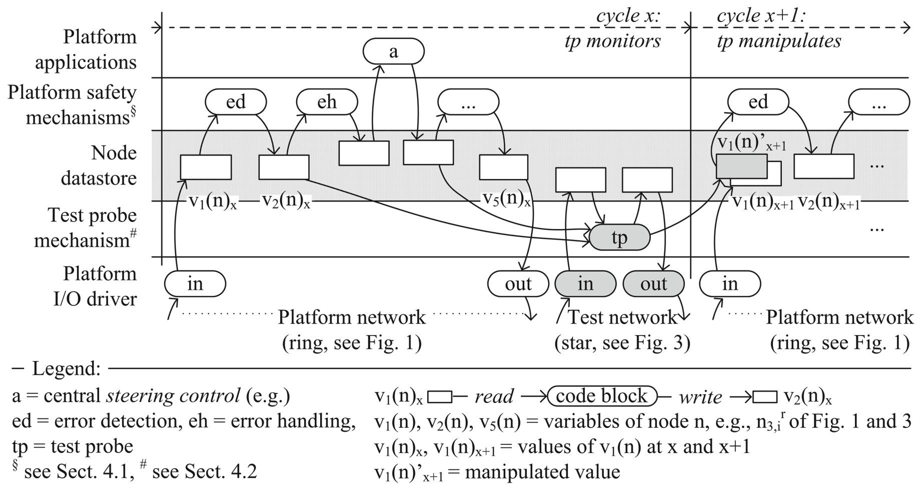

Platform applications
Platform safety mechanisms§
Node datastore
Test probe mechanism#
Platform I/O driver
cycle x: tp monitors
cycle x+1: tp manipulates
a
ed
eh
...
tp
in
out
v1(n)x
v2(n)x
v5(n)x
v1(n)'x+1
v1(n)x+1
v2(n)x+1
Platform network (ring, see Fig. 1)
Test network (star, see Fig. 3)
Legend:
a = central steering control (e.g.)
ed = error detection, eh = error handling,
tp = test probe
§ see Sect. 4.1, # see Sect. 4.2
v1(n)x — read → code block — write → v2(n)x
v1(n), v2(n), v5(n) = variables of node n, e.g., n3,i r of Fig. 1 and 3
v1(n)x, v1(n)x+1 = values of v1(n) at x and x+1
v1(n)'x+1 = manipulated value

Fig. 2. Data flow in a node with built-in test probe

values, communication packets, node and platform states and quality indicators, for one cycle and for every cycle anew.

Test probe. Each node contains a built-in test probe which is a platform test service. Test probe operations are always scheduled at the very end of every cycle. In this position, a test probe can (1) monitor data accumulated in the data store during the last cycle (in Fig. 2: *cycle x*) and (2) manipulate data for the next cycle (in Fig. 2, *cycle x+1*).

Exclusive test resources. Test probes use exclusive time slots (CPU times), memory areas, and access points to a separate test network. Time, space and bandwidth available to a test probe are set to upper limits, constant across all node cycles. Other mechanisms and applications cannot use resources of a test probe, even when it is deactivated. Otherwise test probes would be intrusive.

5 Safety Tests

Test requirement R1: *Redundant input signals* (Sect. 3.3). The minimalistic system under test consists of three nodes: two sensor nodes determine the position of the steering wheel (in Fig. 1: n_1^r, n_2^b) and provide redundant steering angles to any central node (in Fig. 1: one of $\{n_{3,i}^r, n_{3,j}^r, n_{4,i}^b, n_{4,j}^b\}$) that hosts ($n = H(i)$) any instance ($i$) of the central *steering control* application ($i \in I(sc)$, enumerated in Sect. 3.2). The test idea is to manipulate the output of a sensor with different values for different time periods so that the central node must assume that the sensor has a temporary or permanent problem. While one sensor fails temporarily or permanently (controlled by different test data vectors), the central *steering control* shall obtain steering angles from the redundant, error-free sensor.

Test 1. Tolerate failing sensor

```
 1: TEST Tolerate failing sensor WHAT 2 redundant sensors WHEN 1 sensor fails WITH
 2:     N1,a, N2,N3, // Control computer node N1, periphery nodes N2,N3 (sensors), application a
 3:     M, vM, vMx, // Node M where value vMx is injected into variable vM
 4:     c, cc, // Injection instant (cycle c) and injection duration (number of cycles cc)
 5:     t // Delta across two succeeding sensor (signal) values
 6: EXPECT Application continuously receives correct signal values
 7: PROVIDED THAT // System predicates checking the applicability of the test to the target
 8:     IsIn(DN, N1) AND IsIn(PN, N2, N3) AND N2 $\neq$ N3 // Sets DN and PN as def. in Sect. 2
 9:     IsHost(N1, a) // Node N1 executes an instance of application a as def. in Sect. 2
10:     IsIn(V(M), vM) // Set V(M) of the names of all variables of node M as def. in Sect. 2
11:     IsIn({N1, N2, N3}, M) // Via node M faults can be injected into one of N1, N2 or N3
12: CYCLE LENGTH 10 // Specified in milliseconds e.g., 10
13: MAX CYCLES 100 // Obtain definite verdicts within a maximum of 100 cycles
14: SETUP Tolerate failing sensor WITH N1, N2, N3 // Setup of target system (sys. under test)
15: START eNormal == N*.State // Start test clock after SETUP when all nodes operate normally
16: INVARIANT // Safety property (R1.1, Sect. 3.3) must hold in each test clock cycle
17:     N1.a.In == N1.a.In@[-1] DELTA t // In each test cycle compare current with former value
18: CYCLE // Test clock cycles
19:     FROM c TO c + cc - 1 DO M.vM = vMx // Inject value vMx into variable vM of node M
20: STOP
```

Test 2. Tolerate failing master host

```
 1: TEST Tolerate failing master WHAT 1 master and 1 slave WHEN Master fails WITH
 2:     N1i, N1j, N2i, N2j, a, // Dual nodes $dn_1, dn_2$ executing 4 instances of application a
 3:     vN1, vN1ix, vN1jx, // Inject values v1N1ix and v1N1jx in variables vN1 of N1i and N1j
 4:     c, cc, // Injection instant (cycle c) and injection duration (number of cycles cc)
 5:     d, // Number of cycles for the slave to detect the failed master (SH, MH in Sect. 2)
 6:     s // Number of cycles for switching the master
 7: EXPECT Slave becomes master in time
 8: PROVIDED THAT // System predicates checking the applicability of the test to the target
 9:     IsIn(DN, N1i, N1j, N2i, N2j) // Set DN as def. in Sect. 2
10:     IsDN(N1i, N1j) AND IsDN(N2i, N2j) AND N1i$\neq$N2i // Test for 2 different dual nodes
11:     IsIn(V(N1i),vN1) AND IsIn(V(N1j),vN1) // Sets of variables V(N1i), V(N1j) as in Sect. 2
12:     vN1ix $\neq$ vN1jx AND d > 0 AND s > 0
13: CYCLE LENGTH 10 // Specified in milliseconds e.g., 10
14: MAX CYCLES 100 // Obtain definite verdicts within a maximum of 100 cycles
15: CONDITIONS // System predicates which can change values during runtime
16:     IsMH(Ni, Nj, A): eMaster == Ni.A.Authority AND eMaster == Nj.A.Authority
17:     IsSH(Ni, Nj, A): eSlave == Ni.A.Authority AND eSlave == Nj.A.Authority
18: SETUP Tolerate failing master WITH // Setup of target system (system under test)
19:     N1i, N1j, Delay1 = 0, // $dn_1$ starts with no delay to become the master
20:     N2i, N2j, Delay2 = 10, // $dn_2$ starts with 10 cycles delay to become the slave
21:     StateExpected = eNormal // Setup finished when all nodes operate normally
22: START IsMH(N1i, N1j) // Start test clock after SETUP, when $dn_1$ is master.
23: INVARIANT // Safety property (R2.1 and R2.4, Sect. 3.4) must hold in each test clock cycle
24:     (IsMH(N1i, N1j, a) XOR IsMH(N2i, N2j, a)) OR IsSH(N1i, N1j, a) OR IsSH(N2i, N2j, a)
25: CYCLE // Test clock cycles
26:     FROM 0 TO c - 1 DO IsMH(N1i, N1j, a) // Master of $a$ is $dn_1$ because of starting earlier
27:     FROM c TO c + cc - 1 DO N1i.vN1 = vN1ix; N1j.vN1 = vN1jx // Break the master
28:     FROM c + d + s DO IsMH(N2i, N2j, a) // Master of $a$ is $dn_2$ after master switch
29: STOP
```

Test 1 checks safety property R1.1 throughout a test run as a test invariant (line 17), also while manipulating a variable of one of the nodes under test for cc cycles (line 19). It is not necessary for the test to simulate the environment because, with RACE, nodes can start and run in a neutral mode processing default values. With the steering wheel in neutral position (default), dependable delivery of steering angles to the central *steering control* can be tested with the following test vector:

N1 = $n^r_{3,i}$, N2 = n^r_1, N3 = n^b_2, M = n^r_1, // Nodes of the target system (Fig. 1)
a = SteeringControl, // Corresponds to a_2 in Fig. 1
vM = Out.SteeringAngle, // Corresponds to $v_5(n)$ in Fig. 2 with n = M = N2
vMx = 0xDEAD, t = 1.0, c = 30, cc = 2 // Irregular steering angles for 2 cycles

To test the reaction on permanent sensor failure, we extend the fault injection period cc from 2 to, say, 1000 cycles. For scoping the fault region differently, e.g., when looking for fault reasons with exploratory tests during system maintenance, the test can intervene in the data flow in the central node by manipulating the signal quality attribute on the side of the signal receiver, with all other test vector arguments unchanged, as follows: M = $n^r_{3,i}$, vM = In.SteeringAngle.Error, vMx = eErrorConfirmed. If these tests pass, then we can say that the central *steering control* application is indifferent to the sender of the steering angle (R1.4), as well as to other tested faults.

Test requirement R2: *Fail-operational data processing* (Sect. 3.4). The minimally realistic system under test is a core platform of two dual nodes: $dn_1 = (n^r_{3,i}, n^r_{3,j}), dn_2 = (n^b_{4,i}, n^b_{4,j})$ in Fig. 1. The test idea is to shock the nodes of the master host ($MH(sc)$) of the central *steering control* (sc) application so that ($MH(sc)$) backs out. The slave host ($SH(sc)$) shall become $MH(sc)$ within the required time period, including the time needed for error detection plus the time needed for error recovery (switching from SH to MH).

Test 2 checks safety properties R2.1 and R2.4 (line 24) throughout a test run, also while the test injects (line 27) different values vN1ix and vN1jx in the duplicated variables N1i.vN1 and N1j.vN1 (line 3). The safety mechanisms of both nodes must detect this inconsistency (shock) and switch off the master host. The slave host takes over the master role (line 28) and continues executing the platform application. With the following test vector, Test 2 does not inject a fault into an arbitrary memory cell or I/O buffer. Rather, Test 2 attacks the system under test later in the data flow where the platform's error detection service stores the quality (error) indicator for further processing:

N1i = $n^r_{3,i}$, N1j = $n^r_{3,j}$, N2i = $n^b_{4,i}$, N2j = $n^b_{4,j}$, // Platform control comp. (Fig. 1)
a = SteeringControl, // Corresponds to a_2 in Fig. 1
vN1 = Twin.ErrorIndicator, vN1ix = 7, vN1jx = 0, c = 10, cc = 1, d = 3, s = 2.

6 Plausibility Check and Test Analysis

Safety tests written in ALFHA provide reliable statements on system behavior without probe effects, because: (1) Target systems are designed for testability with lifelong built-in test probes (special modules) and data stores decoupling modules (code blocks), see Sect. 4.2; (2) Accurate and understood tests are written in an appropriate domain-specific language that describes fault-injection tests of testable target systems, see Sect. 5; (3) A test system with a central test controller is decoupled from target systems via test probes and separate test networks, see Sect. 4.2 and sketched in Fig. 3; (4) Traces produced by the test controller enable plausibility checks, e.g., Trace 1 for a *tolerate failing master host* test (Test 2) of the target system RACE in operation, see Fig. 4.

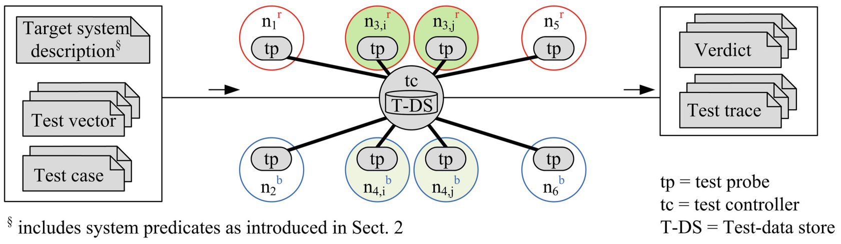
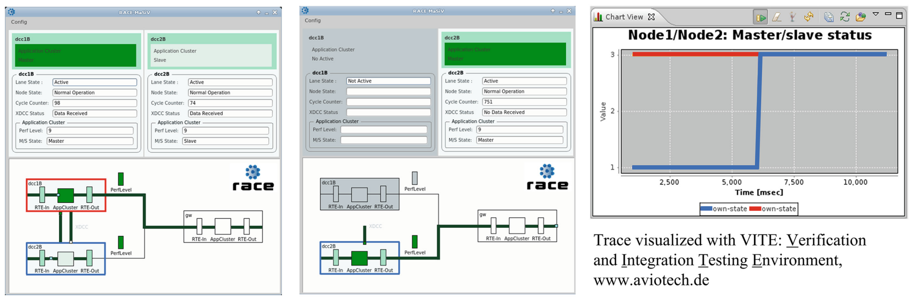


Fig. 3. Test system connected to the target system (Fig. 1) by a separate star network (Color figure online)



RACE-MaSiV
Config

	dcc1B	dcc2B
Application Cluster	Master	Slave
Lane State :	Active	Active
Node State:	Normal Operation	Normal Operation
Cycle Counter:	98	74
XDCC Status	Data Received	Data Received
Perf Level:	9	9
M/S State:	Master	Slave

RACE-MaSiV
Config

	dcc1B	dcc2B
Application Cluster	No Active	Master
Lane State :	Not Active	Active
Node State:		Normal Operation
Cycle Counter:		751
XDCC Status		No Data Received
Perf Level:		9
M/S State:		Master

Trace visualized with VITE: Verification and Integration Testing Environment, www.aviotech.de

a) dn_1 (dcc1B) is master (dark green, top left), dn_2 (dcc2B) is slave (light green)

b) dn_2 (dcc2B) is master (dark green, top right), dn_1 failed (gray)

c) dn_2 changes from slave (status 1) to master (status 3), dn_1 failed (status 3 to not active)

Fig. 4. Demonstration of the *Tolerate failing master host* test (Color figure online)

Once the test system and a testable target system are set up and connected, the focus of plausibility checks moves to analyses of tests and related traces. Test traces document the bindings between test (vector) arguments and parameters of test procedures (Trace 1, lines 1–9) on the basis of structural descriptions of target systems. The test controller uses target system descriptions also for checking whether test cases can be applied to target systems before test runs (e.g., Test 2, lines 9–11). In Trace 1 the test controller documents that in platform cycle 298 test probes of $n^r_{3,i}$ and $n^r_{3,j}$ are instructed to manipulate variables C ($n^r_{3,i}.Twin.ErrorIndicator$) and G ($n^r_{3,j}.Twin.ErrorIndicator$) in platform cycle 308. Trace 1 filtered (1) for values that test probes send to the test controller (gray lines, platform cycles 299, 300, ..., 309, ..., 398) and (2) for values of variables that indicate the role of a dual node (green boxes, variables D and H for dual node dn_1, L and P for dual node dn_2, MH = 3, SH = 1) shows that the target system (platform control computer in Fig. 1) satisfies requirement R2 (Sect. 3.4) for this test run. Snapshots of a RACE-specific trace visualizer (Fig. 4) for another *tolerate failing master host* test of a RACE system can be mapped to Trace 1 as follows: snapshot (a) corresponds to, e.g., platform cycle 308 (test cycle 9) and snapshot (b) corresponds to, e.g., platform cycle 398.

Test Trace 1 Trace of a *Tolerate failing master host* test

```
 1: T-DS 1.1 :A: n^r_{3,i}.Cycle                       # N1i = n^r_{3,i}, Node cycle in scope by default
 2: T-DS 1.8 :B: n^r_{3,i}.State                       # N1i = n^r_{3,i}, Node state in scope by default
 3: T-DS 1.23:C: n^r_{3,i}.Twin.ErrorIndicator         # N1i = n^r_{3,i}, vN1 = Twin.ErrorIndicator
 4: T-DS 1.44:D: n^r_{3,i}.SteeringControl.Authority   # N1i = n^r_{3,i}, a = SteeringControl
 5: ...
 6: T-DS 2.23:G: n^r_{3,j}.Twin.ErrorIndicator         # N1j = n^r_{3,j}, vN1 = Twin.ErrorIndicator
 7: ...
 8: T-DS 3.44:L: n^b_{4,i}.SteeringControl.Authority   # N2i = n^b_{4,i}, a = SteeringControl
 9: ...
10:              #:  A:B:C:D:  E:F:G:H:  I:J:K:L:  M:N:O:P:
11:              #==========================================
12: ...
13: CYCLE:298:  :>:   : :7: :   : :0: :   : : : :   : : : : @9=1 // Controller tc in Fig. 3
14:   instructs n^r_{3,i}, n^r_{3,j} to manipulate C, G in test cycle 9 for 1 cycle (see line 24 below)
15: CYCLE:299: 0:<:299:3:0:3:299:3:0:3:289:3:0:1:289:3:0:1: // Monitor A, B, ..., P
16: CYCLE:299: 0:w:   : : :3:   : : :3:   : : :1:   : : :1: %24 // Invariant holds, line 24
17: CYCLE:299: 0:v:   : : :3:   : : :3:   : : : :   : : : : %26 // IsMH passes, line 26
18: CYCLE:300: 1:<:300:3:0:3:300:3:0:3:290:3:0:1:290:3:0:1: // Monitor A, B, ..., P
19: CYCLE:300: 1:w:   : : :3:   : : :3:   : : :1:   : : :1: %24 // Invariant holds, line 24
20: ...
21: CYCLE:308: 9:<:308:3:0:3:308:3:0:3:298:3:0:1:298:3:0:1: // Monitor A, B, ..., P
22: CYCLE:308: 9:w:   : : :3:   : : :3:   : : :1:   : : :1: %24 // Invariant holds, line 24
23: CYCLE:308: 9:v:   : : :3:   : : :3:   : : : :   : : : : %26 // IsMH passes, line 26
24: CYCLE:308: 9:c:   : :7: :   : :0: :   : : : :   : : : : %27 // tp manipulate C and G
25: CYCLE:309:10:<:   : : : :   : : : :299:3:0:1:299:3:0:1: // Monitor I, J, ..., P
26: CYCLE:309:10:w:   : : : :   : : : :   : : :1:   : : :1: %24 // Invariant holds, line 24
27: ...
28: CYCLE:398:99:<:   : : : :   : : : :388:3:0:3:388:3:0:3: // Monitor I, J, ..., P
29: CYCLE:398:99:w:   : : : :   : : : :   : : :3:   : : :3: %24 // Invariant holds, line 24
30: CYCLE:398:99:v:   : : : :   : : : :   : : :3:   : : :3: %28 // IsMH passes, line 28
31: VERD 0       #========================================== // Test passes (0: no errors)
```

7 Summary

The tests presented in this paper demonstrated a method of proving safety-related statements about a fault-tolerant system, like “a steer-by-wire car remains steerable when one computer of the central platform computer fails.” More fault-injection tests for the same target at different points of attack (e.g., nodes and variables) and in different situations (e.g., degradation modes and load levels) are necessary to increase the confidence in and precision of such statements. Test probes permanently built into all nodes of a fault-tolerant, cyber-physical system that executes time-controlled behavior provide the necessary testability.

References

1. Ayestaran, I., et al.: Modeling and simulated fault injection for time-triggered safety-critical embedded systems. In: 2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pp. 180–187, June 2014
2. Becker, K., et al.: RACE RTE: a runtime environment for robust fault-tolerant vehicle functions. In: 11th European Dependable Computing Conference on CARS Workshop - Dependability in Practice. IEEE, September 2015
3. Bondavalli, A., Simoncini, L.: Failure classification with respect to detection. In: Proceedings of 2nd IEEE Workshop on Future Trends of Distributed Computing Systems, 1990, pp. 47–53, September 1990
4. Büchel, M., et al.: An automated electric vehicle prototype showing new trends in automotive architectures. In: International Conference on Intelligent Transportation Systems (ITSC 2015). IEEE, September 2015
5. Frtunikj, J., et al.: Qualitative evaluation of fault hypotheses with non-intrusive fault injection. In: 5th International Workshop on Software Certification (WoSoCer 2015). IEEE, November 2015
6. Kane, A., Fuhrman, T., Koopman, P.: Monitor based oracles for cyber-physical system testing : practical experience report. In: 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 148–155, June 2014
7. Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications. Springer, New York (2011)
8. Sommer, S., et al.: RACE: a centralized platform computer based architecture for automotive applications. In: Vehicular Electronics Conference and the International Electric Vehicle Conference (VEC/IEVC). IEEE, October 2013
9. Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: MODIFI: a MODel-implemented fault injection tool. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 210–222. Springer, Heidelberg (2010)