Qualitative Evaluation of Fault Hypotheses with Non-Intrusive Fault Injection

Jelena Frtunikj
fortiss GmbH
Guerickestraße 25
80805 München, Germany

Joachim Fröhlich
Siemens AG
Otto Hahn Ring 6
81739 München, Germany

Tim Rohlfs
jambit GmbH
Erika-Mann-Straße 63
80636 München, Germany

Alois Knoll
Technische Universität München
Boltzmannstraße 3
85748 Garching, Germany

***Abstract*—This paper presents a new approach for demonstrating whether safety-critical, hard real-time systems implement fault hypotheses correctly and timely. In the forefront are tests which non-intrusively and deterministically stimulate and monitor the system under test. The tests use a domain-specific language which can formalize logical truths on system properties derived from fault hypotheses. Test results are strong arguments in safety cases. In this way the tests support both development and certification of safety-critical systems. Advantages over existing approaches to evaluating safety properties of complex and diverse safety-critical systems are discussed briefly, and fundamental work is referenced.**

***Keywords*—safety, safety properties, safety case, fault hypothesis, fault injection test**

I. Introduction

The evaluation of safety properties of safety-critical systems and related fault-hypotheses is a challenging task. Of all safety-critical systems fail-operational systems have to meet the highest availability and reliability requirements. In order to dependably provide safety mechanisms in typical and critical situations, fail-operational systems with extensive computer hardware and software are realized as distributed systems consisting of redundant components, replicated communication channels and redundancy control so as to tolerate transient and permanent faults without bringing down the complete system. To ensure that there is evidence that a complex safety-critical system implements safety properties timely and correctly according to the system's fault hypothesis, different functional safety standards impose various requirements on processes and systems and propose methods for implementing and verifying these requirements according to system and context specific safety integrity levels. The recently released ISO 26262 [1] targeting automotive systems represents many similar but nonetheless different functional safety standards. ISO 26262 recommends fault injection tests at various system levels for assessing the effectiveness of the safety mechanisms and demonstrating the correct implementation of safety requirements. This raises the issue of how to inject hardware and software faults into partially or fully integrated systems during tests without affecting the behavior of the systems in any other aspect than is intended.

Safety-critical systems usually run under hard real-time constraints with response times of a few milliseconds even when faults occur or system components fail. This creates special challenges for test systems and tests which check safety properties. A test system must be capable of injecting faults, seed and capture data at the right places (internals of redundant components) and at the right point in time in order to verify detection and error handling mechanisms. To avoid functional and temporal distortions of the system under test, tests must run free of undesirable side-effects, not related to the system stimuli. Otherwise tests would not be sound and hence could not be used in safety cases.

The contributions of the work presented here are twofold: (1) Formalization of fault hypotheses for safety-critical, hard real-time, distributed systems in form of tests with the help of a new, domain-specific test language called ALFHA[1], (2) Specification of ALFHA tests that can seed and monitor signal and state data into integrated system parts and systems in a deterministic, non-intrusive manner. The tests enable qualitative (structural) system analyses.

This article is structured as follows: In Section II, we characterize target systems that enable non-intrusive, fault-injection tests without side-effects. We introduce RACE[2], a safety-critical system from the automotive sector which has the required characteristics of qualified target systems. Later we use RACE for a non-intrusive fault-injection test. Since fault hypotheses are the starting points for the approach presented here, in Section III we define a schema for describing fault hypotheses in terms of testable, temporal truth statements about the behavior of different systems under tests when faults occur. Section IV presents key concepts of the language ALFHA that facilitates tests of these fault hypotheses. Section V outlines an ALFHA test that checks the fail-operational capability of the RACE system. The test system and its integration with a target system are in the focus of Section VI. Section VII argues similarities and differences with related work. The paper finishes with a summary and outlook.

II. Target Systems

Prior to field application, safety-critical systems must demonstrate that they implement the required safety properties timely and correctly according to fault hypotheses derived from safety cases. When using fault injection to demonstrate and evaluate safety properties the question arises: What requirements must a safety-critical system fulfill to enable fault injection tests which permit dependable and deterministic statements on the system's reliability?

[1] Assertion Language for Fault-Hypothesis Arguments

[2] Robust and reliant Automotive Computing environment for future Ecars, www.projekt-race.de/en

The class of safety-critical systems considered in this paper operate under tight and hard real-time constraints. Ideally, critical system functions are scheduled in a time-triggered manner under control of the internal clock rather than in an event-triggered manner where external incidents can stress the system. This provides a safety-critical system a more deterministic behavior for the mission for which it is constructed and dimensioned. Fast time-triggered systems typically progress in cycles of milliseconds or multiples thereof. The system by design contains a module that enables injection of faults at exact locations and cycles. This module, hereafter referred to as test probe, operates in a time-triggered manner like all other system modules. Test probes run at the end, and therefore at the beginning, of every cycle. In this position between two adjacent cycles, test probes are enabled to (a) monitor system data accumulated in the last cycle and to (b) manipulate system data for the next cycle. The system accumulates data per cycle and in different processing stages in a real-time system database. Distributed target systems contain a real-time system database and built-in test probe in each system node. Each real-time system database captures system-relevant data-flows within and among system modules and system nodes (Figure 1).

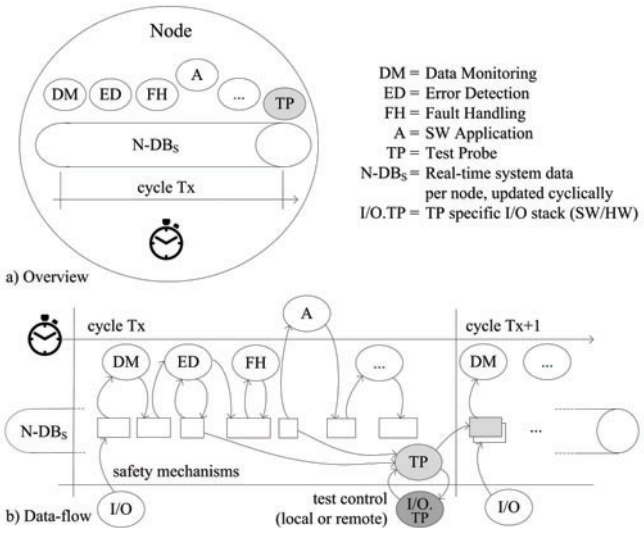

Fig. 1. Data flow in a time-triggered system with built-in test probes

Reliable fault-injection tests provide identical results when applied to the system in the lab or in the field, under comparable conditions. Hence, in all circumstances, fault injection tests must run free of side-effects other than the fault itself in order to avoid functional and temporal distortions of the system under test. Test probes acting like any other module from the point of view of the system scheduler is one measure of this kind. Exclusively reserved time slots, memory areas and network bandwidth for test probes is another measure.

Furthermore, each test probe is connected to a test control machine via a point-to-multipoint connection physically separated from the transmission medium that the system nodes use for communication. When scheduled at the end of a system cycle a test probe polls a dedicated test I/O port (Figure 1.b, I/O.TP). Each test probe and the test control machine exchange maximum one frame per cycle and direction, limiting the WCET of test probe operations. Other modules cannot use time slots, memory area and the I/O port of a test probe when no tests run. Otherwise a probe would be intrusive and the behavior of the system in the field would differ from the behavior of the tested system in the lab.

The safety critical, distributed system RACE [5], [12] is an almost ideal subject for qualitative evaluation of fault hypotheses based on non-intrusive fault injection. In particular, RACE stays operational when redundant electric circuits, sensors, actuators or central processors fail. Hence, we use RACE for stating and evaluating fault hypotheses on fail-operational behavior. On every system node RACE offers safety mechanisms as part the run-time environment (RTE) between low-level, redundant hardware and high-level system functions, like steering, braking and accelerating. In RACE, the central processor is physically divided into several duplex control computers (DCC) connected by a bidirectional communication ring. A DCC consists of two execution channels. Both channels monitor input and output data mutually. In case of a channel inconsistency, the faulty DCC backs out to not jeopardize the operation of a RACE system. A safety mechanism guarantees fail-operational behavior when a redundant DCC takes over system control tasks from a failing DCC. Moreover, aggregate (sensor and actuator) redundancy is also handled by the RACE RTE (platform) using voting mechanisms. Steering a car is an example of a safety-critical system function that utilizes the safety mechanisms of the RACE platform in general and the central processor in particular. In RACE the system function *steering* consists of three RTE applications running on three different RACE nodes: one RTE application runs on the steering wheel aggregate (front-end), one runs on the steering actuator (back-end), and the steering logic runs in between on the central processor. Section V illustrates a fault-injection test for the central processor of RACE running a steering application. Beforehand, the fault hypotheses schema (Section III) and a formal language for automatic tests of fault-hypotheses (Section IV) are introduced.

III. Fault Hypotheses Schema

Fault hypotheses drive the development and verification of safety-critical systems. When developing safety critical systems the selection of safety mechanisms and their implementation depend upon the underlying fault hypotheses. A fault hypothesis in turn is derived from the system safety analysis. Since we want to obtain deterministic statements about the effectiveness of safety mechanisms from test runs, we need fault hypotheses that describe qualitative system properties. Hence, we assume that fault hypotheses describe types, number and sequences of system-internal or system-external faults that a safety-critical system shall detect, handle and, in the strongest form, tolerate. For a safety analysis requiring a given probability of failure, a fault hypothesis must address the following seven points in order to cover failure relevant properties if one or more triggering faults will occur:

(1) Fault tolerance region (FTR). The minimal subsystem of the target system that tolerates failures of one or more components constituting the FTR.

(2) Fault containment region (FCR). The maximal subsystem of the target system that operates correctly regardless

of faults outside the FCR. This holds for faults captured in the fault model of the system.

(3) Failure mode. The type of behavior that the FTR expects of an FCR impacted by a fault. The behavior ranges from very restricted (fail-silent) to unrestricted (undefined).

(4) Failure persistence. Failures are classified according to the temporal persistence. Permanent failures prevail until repair, transient failures disappear without an explicit repair, and intermittent failures usually appear as irregular sequences of transient failures of degraded devices.

(5) Error detection latency. The time it takes the target system to detect an error which is a part of the system state liable to lead to a subsequent failure.

(6) Recovery interval. The time it takes the target system to recover from a fault.

(7) Safety mechanism. A mechanism that the target system uses or implements to detect faults or control component failures in order to achieve or maintain a safe state.

Altogether these points characterize the fault-specific structures (regions) and behavior of a safety-critical system facing a single fault or multiple faults which can lead to a system failure if not properly handled. One or more fault hypotheses can be stated for a safety-critical system. Each fault-hypothesis is automatically checked with a test suite consisting of one or more test cases formalized with ALFHA (Section IV) as Section V demonstrates.

IV. Assertion Language for Fault Hypotheses

Tests checking fault hypotheses in the sense of Section III under hard real-time constraints must be able to stimulate the system under test with faults at the right time, without impairing the behavior of the system under test in any way. The test language ALFHA is aimed at distributed systems with redundant time-triggered components, where triggers are fired in discrete multiples of a base rate measured in milliseconds. ALFHA offers constructs for specifying space-time coordinate points in the system under test. At these points ALFHA tests seed signal and state data or capture data on a per cycle basis for comparing and checking properties within defined regions, for example within FTRs or FCRs. The issue of ALFHA tests avoiding undesired and hence defective side-effects is addressed in Section II and Section VI.

Procedures, parameters and variables. ALFHA aims at, among other things, separation and combination of tests of safety elements out of context as well as in context of different systems and situations. To this end, ALFHA separates parameterizable test setup procedures (SETUP...WITH...) from parameterizable test procedures (TEST...WITH...). These procedures can be combined and used in different contexts just by calling them with different sets of parameter arguments describing target systems, system stimuli and context dependent expectations. The consistency of arguments passed as procedure parameters and variables are checked before test runs as part of static test applicability checks (RESTRICTIONS).

Test actions, system variables and node variables. Test setup procedures and test procedures contain test actions. Test actions in a test setup procedure bring the system under test into a defined state where the test procedure starts. Test actions operate either with system variables or with node variables. System variables designate system nodes. They are special insofar as target systems are, in many cases, distributed systems of varying numbers of connected nodes with different tasks. All other variables are node variables. Each node variable belongs to exactly one system node. START, STOP, CONTINUE are test actions that can only be applied to system nodes. Monitor (ASSERT, ==) and manipulate (=) are applied to node variables. Invariants are asserts which shall hold during a complete test run. At call-up point (cycle) a test condition of an invariant must evaluate to true (!!, strong invariant) or can temporarily be undefined (!, weak invariant). They must never evaluate to false. Test conditions of asserts must evaluate to true or to false.

Test clocks and clock conditions. Target systems utilize time-triggered architectures and hence run cyclically. So do the test system and hence ALFHA tests with an important difference. The pace of test clocks and the duration of the test cycles are determined by the clocks of all nodes of the system under test. The test system signals a timing issue if these clocks tick irregularly or drifts intolerably. Furthermore, ALFHA distinguishes absolute clocks and relative clocks. An absolute clock ticks throughout a test run. The fastest booting node of the target system starts the absolute clock. However, in the strict sense, a test effectively starts when the system under test satisfies clock start conditions (CLOCK WHEN...) triggering test relative clocks.

Time points and phases. ALFHA tests can provide precise statements on time-dependent behavior of target systems because in ALFHA exact cycles as well as phases (several succeeding cycles) can accurately be specified within the time granularity SYSTEM PERIOD. ALFHA tests execute test actions within a cycle ([Tx]) or during a bounded sequence of cycles ([Tx:Ty]) of either relative clocks in test procedures or absolute clocks in test setup procedures. Each test run is limited in time in order to obtain a definite test verdict.

These constructs form the backbone of an ALFHA test. Whether ALFHA tests can falsify or confirm fault hypotheses primarily depends on the richness of the data model of a target system, i.e., on what system variables and node variables can be monitored and manipulated, what they represent and how they relate. The data model is derived from the fault model as are the fault hypotheses.

V. Example of a Fault Hypothesis Test

This section explains a fault hypothesis and a corresponding ALFHA test that evaluates a fail-operational property of RACE. The core of a system built with RACE is a platform providing safety mechanisms to RACE applications (system functions). The test investigates two duplex control computers (DCCs) of a RACE platform connected by a bidirectional ring and supplied with electricity by two separate circuits (Figure 2). Every DCC comprises two channels, which monitor each other while executing the same control tasks. Hence, the target system under test in this case is distributed on four central nodes. The *master switch* safety mechanism built into the RACE platform enables platform applications to stay operational in case of the DCC acting as the master of an

application fails. The *master switch* mechanism running on both DCCs detects the failing master. The slave DCC takes over the master role and continues executing the platform application (Table I).

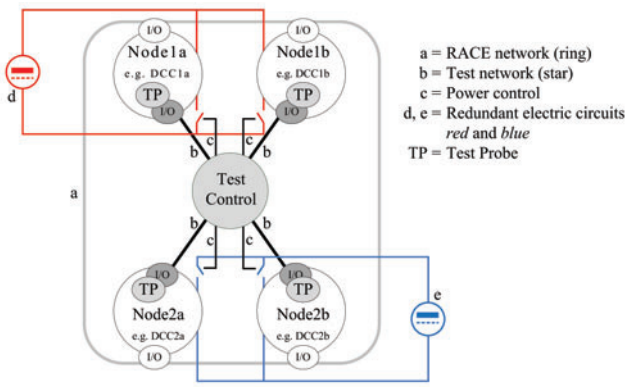

Fig. 2. Topology of test system and an example target system

Failure derived from the fault model	Redundant duplex control computer (DCC) fails.
Fault tolerance region	Vehicle control computer (VCC) composed of 2 DCC including the RTE, forms an FTR.
Fault containment region	Every DCC consisting of two channels each, including the RTE and the applications of concern, represent an FCR.
Failure mode	Application running on the VCC stays operational when the master DCC hosting the application fails.
Failure persistence	The DCC executing the application as master fails permanently.
Error detection latency	Slave DCC detects failure of master DCC within 10 milliseconds.
Recovery interval	Slave DCC becomes master 10 milliseconds after error detection.
Safety mechanism	Distributed master switch algorithm evaluates DCC states and controls redundant execution of application on several DCC.

TABLE I. FAULT HYPOTHESIS A: MASTER SWITCH

The *master switch* test (Algorithm 1) is parametrized with four RACE nodes (Algorithm 1, lines 6-7). These four nodes constitute two DCCs. The separate test network links their test probes (TP) to the central test control machine (ALFHA processor, see also Figure 2 and Figure 3). To induce different channel inconsistencies in the master DCC during different test scenarios, the test takes additional parameters indicating the memory locations in both channels that the test shall manipulate (Algorithm 1, lines 8-9).

Before execution the test checks whether it can be applied for the test arguments given. For example, the test assumes four unique nodes, which in pairs of two channels build a central platform of two DCC (Algorithm 1, lines 16-18). If applicable to the target system, the test switches on the four nodes (Algorithm 1, lines 26-30) via relay cards (Figure 2, c) and waits until they are in a steady system state (Algorithm 2, line 14). The main part of the tests starts as soon as the system satisfies the clock condition (Algorithm 1, line 31). Remaining test actions are triggered by ticks of the relative clock.

When two or more computers redundantly execute a control program (RACE application) then, for sake of unambiguous control, at each time point only one of these programs is allowed to send signals to the connected actuators which translates to: only one program of a group of redundant programs acts as master (Algorithm 1, line 32):

INVARIANT $Node1IsMaster$ XOR $Node2IsMaster$

The *master switch* test procedure checks this safety property in each test cycle. In this example we suppose that, for acceptably short periods, no master DCC is available in the target system. Accordingly, the safety property is relaxed permitting exactly one master DCC down period for a given number of cycles (Algorithm 1, line 41, weak invariant (!)), which shall be at most two cycles according to Table I.

In line with the weakened safety property, the *master switch* test shall be called for a target system sketched in Figure 2 with the following arguments (see also Test Trace 1, line 2):

$Application = Steering$,
$Node1a = DCC1a$, $Node1b = DCC1b$,
$Node2a = DCC2a$, $Node2b = DCC2b$,
$Var1a$ = $TwinChannel.ErrorIndicator$, $Val1a = 7$,
$Var1b$ = $TwinChannel.ErrorIndicator$, $Val1b = 0$,
$InjectionCycle = 10, InjectionDuration = 3$,
$SwitchCycles = 2, CycleLength = 10, MaxCycles = 100$

Actually, this test case does not inject a fault into an arbitrary memory cell, I/O buffer or CPU register. Rather, the test attacks the target system at a different position later in data flow where the RTE stores the evaluation result for further processing. Disregarding the current physical condition, the test overwrites the $ErrorIndicator$ in both nodes of DCC1 with different values 7 (fatal error in channel 1a) and 0 (no error in channel 1b) in order to definitely induce an inconsistency. In line with the fault hypothesis the test expects both DCCs to detect and confirm the inconsistency. The master DCC is expected to back out of the system and the slave DCC shall take over the master role. When repeated or executed for various configurations of the target system and test vectors, the *master switch* test is expected to deterministically demonstrate the fail-operational capability of a central RACE platform.

VI. TEST SYSTEM

Because of aiming at systems running under tight and hard real-time constraints ALFHA tests and the test system must not accidentally change the behavior of the system under test. To this end, ALFHA and the test system rely on test probes built into each node of the (distributed) system under test (Section II). Test probes and related concepts, like time-triggered scheduling and data flow across a real-time database per node, while necessary for avoiding problems with inadmissible side-effects on system run-time and memory usage, do not suffice for dependable fault-injection tests. It is also necessary to efficiently run ALFHA tests without impairing the timing behavior of the target system. This is the responsibility of the test system as a whole, including the central ALFHA processor of the test control machine, the test probes on each node, local ALFHA processors on their own, and the communication infrastructure between the test control machine and the test probes (Figure 2, Figure 3).

Although showing characteristics of a dynamic script language such as wild cards for variable names, ALFHA tests

Algorithm 1 Test *master switch* mechanism

TEST Master Switch
  WHAT Vehicle control computer consisting of 2 duplex control computer **WHEN** Application master fails
  EXPECT Application slave becomes master in time
WITH
  $Application$, // Part of a safety-critical system function, e.g. steering or braking
  $Node1a$, $Node1b$, // Duplex control computer 1 (DCC1)
  $Node2a$, $Node2b$, // Duplex control computer 2 (DCC2)
  $Var1a$, $Var1b$, // Where to inject the fault in Node1a, Node1b?
  $Val1a$, $Val1b$, // What fault to inject in Node1a, Node1b?
  $InjectionDuration$, // Number of cycles to inject the fault
  $SwitchCycles$, // Maximum number of cycles allowed for switching an application's master
  $InjectionCycle$, // When to inject the variable values?
  $CycleLength$, // The length of a period (cycle) in milliseconds, e.g., 10 milliseconds
  $MaxCycles$ // Limit the test run to obtain a definite verdict
RESTRICTIONS
  $IsUnique(Node1a, Node1b, Node2a, Node2b)$
  $AreTwins(Node1a, Node1b)$
  $AreTwins(Node2a, Node2b)$
  $SwitchCycles > 0$
  ... // e.g. $Val1a$!= $Val1b$, $InjectionCycle < MaxCycles - InjectionDuration - SwitchCycles$
SYSTEM PERIOD $CycleLength$ **TOLERANCE** 0.1
TIME BOUND $MaxCycles$
CONDITIONS
  $Node1IsMaster : eMaster == Node1*.Run.Application.Authority$
  $Node2IsMaster : eMaster == Node2*.Run.Application.Authority$
SETUP Master Switch **WITH**
  $Node11 = Node1a, Node12 = Node1b, StartDelay1 = 0$, // DCC1 starts with no delay
  $Node21 = Node2a, Node22 = Node2b, StartDelay2 = 10$, // DCC2 starts with 10 cycles delay
  $UpAndRunning = 300$, // Allow the target system to settle
  $StateExpected = eNormalOperation$
CLOCK WHEN $Node1IsMaster$ // Because DCC1 starts before DCC2
INVARIANT $Node1IsMaster$ **XOR** $Node2IsMaster$ // Safety property
BEGIN
  /// 1. A system with two DDCs must determine one master and one slave
  $[: < InjectionCycle]$!! // Strong invariant (!!)
  /// 2. In fault detection phase an inconsistent master is tolerated
  $[: < + InjectionDuration]$!!
    $Node1a.Var1a = Val1a$
    $Node1b.Var1b = Val1b$
  /// 3. During a master switch the system is allowed to run without any master
  $[: < + SwitchCycles]$! // Weak invariant (!)
  /// 4. The faulty master can stay down or come back as slave but the former slave must become the new master
  $[:]$! $Node2IsMaster$
  END
END

Algorithm 2 Setup harness for *master switch* test

SETUP Master Switch
  WHAT 4 nodes of 2 DCCs **WHEN** DCCs start with a delay
  EXPECT DCCs reach expected state
WITH
  $Node11$, $Node12$, // Node pair of DCC1 to start
  $StartDelay1$, // Cycle when DCC1 starts
  $Node21$, $Node22$, // Node pair of DCC2 to start
  $StartDelay2$, // Cycle when DCC2 starts
  $UpAndRunning$, // Cycle when setup is finished
  $StateExpected$ // State of all nodes after setup
BEGIN
  $[StartDelay1]$ $Node11; Node12$ // Power up DCC1
  $[StartDelay2]$ $Node21; Node22$ // Power up DCC2
  $[UpAndRunning]$ $Node*.State == StateExpected$
END

must be compiled for performance reasons. The ALFHA compiler generates a binary test plan from the test procedure, test setup, test data vector and the symbol table identifying all variables in the real-time databases of all nodes as sketched below:

(1) Compiler: Locate and map variables. The compiler expands variables and parameters given as wild cards to individual variables. The compiler then maps symbolic names of all individual variables to nodes of the distributed target system and unique physical addresses by means of the system's symbol table. Thereby the compiler indicates variables and parameters that can not be localized in the target system.

(2) Compiler: Plan actions per cycle, variable and node. The compiler generates a two-dimensional table with each table element codifying the test action (command), such as monitor or manipulate, to be performed in the selected cycle (row) for the selected variable (column). All variables of each

node related to the same test action that shall be executed in the same cycles are combined, for example all variables of node N to monitor from cycle Tx onwards. The compiler minimizes the number of commands per cycle and per node by, for example, compiling a new monitor command only if the set of node variables to monitor changes. The compiler also indicates contradictory test actions and test actions with too many operands (variables, parameters) if they exceed worst case execution times allowed for test probes or for the central ALFHA processor.

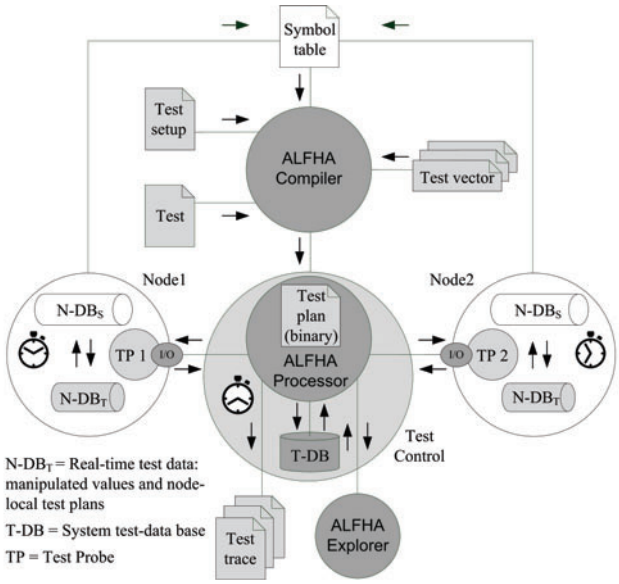

Fig. 3. Data flow in the test system

During testing, the ALFHA processor steps through the compiled test plan, cycle-by-cycle. At the same time the central ALFHA processor processes values of node variables and health data received from all connected nodes in every cycle.

(1) Processor: Perform test actions. Non-empty cells of a compiled test plan contain test actions for the current cycle, i.e., for the current table row. Test actions intended for execution by the central ALFHA processor include: start nodes, start relative test clocks, and check values of variables of more than one node (global asserts). Test actions intended for execution by test probes include monitor and manipulate commands that test probes shall perform autonomously, such as checking values of node variables (local asserts) and incrementing values of node variables on-site. A conditional test action which a test probe can decide locally needs 1 cycle to take effect. A conditional test action which the central ALFHA processor must decide globally needs 4 cycles minimum, including the cycles for data round trip, to become effective. The number of test actions and the number and size of operands (variables and values) an ALFHA processor can process per cycle is reasonably limited, which simultaneously serves time-scheduling restrictions.

(2) Processor: Supervise node vitality. In every cycle, the central ALFHA processor receives from every connected test probe one and only one data packet of constant (maximum) size and processes the transmitted values of node variables in accordance with the test plan, for example data packets from DCC nodes cyclically transferring the master/slave state. Simultaneously, the central ALFHA processor continuously watches out for fatal errors of the test system. Lost or late data packets indicate timing faults. Data packets from a test probe which do not confirm the test action to be performed and data packets of sizes other than the agreed size indicate faults within the test system. Ignoring such faults can invalidate test verdicts.

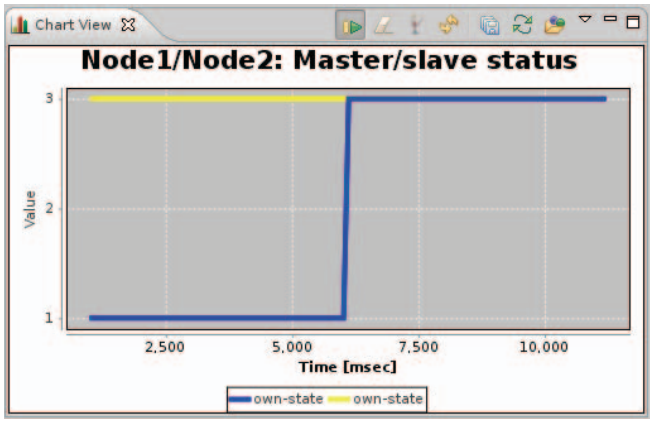

Fig. 4. Visualization of the *master switch* test (Algorithm 1)

The basis for prototyping the ALFHA test system is VITE[3]. Figure 4 was captured with VITE for a trace produced by a *master switch* test of a RACE system with two DCC. The test is implemented with VITE and specified in ALFHA (Algorithm 1). Historic values of the variables $Run.Steering.Authority$ of $DCC1a$ and $DCC2a$ feed the visualization. The interactive front-end of VITE is an instance of an ALFHA explorer (Figure 3). While executing a binary test plan an ALFHA processor traces test actions in tabular form. Test Trace 1 shows a conceived trace for the *master switch* test, derived from and extending VITE traces. Rows tagged with a leading TICK unroll an executed test plan. The unrolled test plan is enriched with actual and manipulated values of signal and state variables of system nodes in scope of the test case and captured in the test-data base (Figure 3, T-DB). ALFHA traces shall explain how the test system produces test results and enable automatic analysis of test runs for false positives and false negatives. For supervising and checking the vitality of target systems and the test system the traces must contain real-time stamps (not considered in Test Trace 1).

VII. Related Work

The verification of fault hypotheses with non-intrusive fault-injection tests as presented in this paper requires systems under test to operate in a time-triggered manner. The time-triggered architecture (TTA) is an approved framework for implementing distributed embedded real-time systems with high dependability requirements [8]. TTA is well suited for implementing target systems as presented in Section II. For example, TTA utilizes a global time base of known precision at every node which enables node clocks of identical cycle lengths ticking with equal pace. Furthermore, TTA defines several node interfaces such as the diagnostic and maintenance

[3]Verification and Integration Testing Environment, www.aviotech.de

Test Trace 1 Partial test trace of a *master switch* test (schema)

```
 1: PROC TEST_Master_Switch
 2: ARGS Application = Steering, Node1A = DCC1a, ..., Var1a = TwinChannel.ErrorIndicator, Val1a = 7, ..., Val1b = 0, ...
 3: PROC SETUP_Master_Switch
 4: ARGS Node11 = Node1a, Node12 = Node1b, StartDelay1 = 0, ...
 5: T-DB 1.1 :A: DCC1a.CycleCounter                                  // System test-data base (T-DB): node 1, variable 1
 6: T-DB 1.8 :B: DCC1a.State                                         // System test-data base (T-DB): node 1, variable 8
 7: T-DB 1.23:C: DCC1a.TwinChannel.ErrorIndicator
 8: T-DB 1.44:D: DCC1a.Run.Steering.Authority                        // Figure 4: DCC1a starts as master (value 3, yellow)
 9: T-DB 2.1 :E: DCC1b.CycleCounter                                  // System test-data base (T-DB): node 2, variable 1
10: T-DB 2.8 :F: DCC1b.State
11: T-DB 2.23:G: DCC1b.TwinChannel.ErrorIndicator
12: T-DB ...
13: T-DB 3.44:L: DCC2a.Run.Steering.Authority                        // Figure 4: DCC2a starts as slave (value 1, blue)
14: T-DB ...
15: #             ========================================
16: #             :  A:B:C:D:  E:F:G:H:  I:J:K:L:  M:N:O:P:          // Map T-DB variables to value traces
17: #             ----------------------------------------
18: TICK SETUP_Master_Switch                                         // Run setup procedure, Algorithm 2
19: TICK ...
20: TICK :300:  :<:298:3:0:3:298:3:0:3:288:3:0:1:288:3:0:1:
21: TICK :300:  :y:   :3: : :   :3: : :   :3: : :   :3: : : %14     // Assert in SETUP_Master_Switch, line 14 succeeds
22: TICK TEST_Master_Switch                                          // Run test procedure, Algorithm 1
23: TICK :300:  :y:   : : :3:   : : :3:   : : :1:   : : :1: %31     // Condition in TEST_Master_Switch, line 31 succeeds
24: TICK :300:  :>:   : :7: :   : :0: :   : : : :   : : : : @10=3   // Manipulate C and G in test cycles 10, 11, 12
25: TICK 1. A system with two DDCs must determine one master and one slave %34 // Documentation comment on line 34
26: TICK :301: 0:<:299:3:0:3:299:3:0:3:289:3:0:1:289:3:0:1:
27: TICK :301: 0:w:   : : :3:   : : :3:   : : :1:    : : :1:%32    // Strong invariant on line 32: completely defined
28: TICK :302: 1:<:300:3:0:3:300:3:0:3:290:3:0:1:290:3:0:1:
29: TICK :302: 1:w:   : : :3:   : : :3:   : : :1:   : : :1: %32     // Strong invariant on line 32: completely defined
30: TICK ...
31: TICK :310: 9:<:308:3:0:3:308:3:0:3:298:3:0:1:298:3:0:1:
32: TICK :310: 9:w:   : : :3:   : : :3:   : : :1:   : : :1: %32     // Strong invariant on line 32: completely defined
33: TICK :310: 9:c:   : :7: :   : :0: :   : : : :   : : : :         // Probes at DCC1a and DCC1b manipulate C and G
34: TICK 2. In fault detection phase an inconsistent master is tolerated %36 // Documentation comment on line 36
35: TICK :311:10:<:309:3:7:3:309:3:0:3:299:3:0:1:299:3:0:1:         // C and G manipulated in test cycle 10
36: TICK :311:10:w:   : : :3:   : : :3:   : : :1:   : : :1: %32     // Strong invariant on line 32: completely defined
37: TICK :311:10:c:   : :7: :   : :0: :   : : : :   : : : :         // Probes at DCC1a and DCC1b manipulate C and G
38: TICK :312:11:<:310:3:7:3:310:3:0:3:300:3:0:1:300:3:0:1:         // C and G manipulated in test cycle 11
39: TICK ...
40: TICK :400:99:<:   : : : :   : : : :387:3:0:3:387:3:0:3:         // DCC1 disappeared, only DCC2 operates
41: TICK :400:99:v:   : : : :   : : : :   : : :3:   : : :3: %32     // Weak Invariant: only DCC2 satisfies conditions
42: TICK :400:99:y:   : : : :   : : : :   : : :3:   : : :3: %43     // Assert on line 43 succeeds: DCC2 is master
43: #             ========================================
44: VERD 0                                                          // Test verdict: no error found.
```

(DM) interface for "setting node internal parameters and for retrieving information about the internals of the node, e.g., for the purpose of internal fault diagnosis. Usually, the DM interface is not time-critical" [8] (p. 118). By comparison, the test probe interface of a target system for ALFHA tests is time-critical by its very nature. Test probes are not intended just for retrieving data for system diagnosis or for setting system parameters before system runs, but also for unobtrusively seeding signal and state data during system runs. In any case, a test probe does not impair the temporal composability of the containing node because it uses exclusively reserved resources in test and production environments. This is in stark contrast to tools like software debugger or profiler, which must consume system resources for instrumentation. While none of the individual techniques for making test probes and fault-injection tests non-intrusive is original, their combination enables the qualitative evaluation of fault hypotheses without undesired side-effects.

Fault hypotheses are essential for the construction of safety cases and hence for the development, verification and validation of safety-critical systems. The fault hypothesis schema (Section III) builds on the schema used by Kopetz [7] and Obermaisser et al. [9] with an important difference: Faults and resulting failures are treated qualitatively, not quantitatively. Statistical statements about failures rates, operational availability and repair times are not the focus of the presented fault hypothesis schema, since it is used to describe how a target system shall handle concrete faults in concrete situations.

ALFHA tests utilize temporal operators for obtaining reliable statements about a behavior of a system under real-time constraints in the presence of rare events such as faults. Temporal specifications and operators are the innate strength of model checkers. The Specification and Assertion Language SALT [3] is a mature language, which utilizes Timed Linear Temporal Logic operators (TLTL) for obtaining statements about real-time properties of systems with strict execution times and

deadlines. As with every model checker SALT statements can be all-quantified and are verified offline for simplified system models, but not for real system implementations. In contrast, ALFHA statements hold for single but deterministic test runs, operate online and thus check the fulfillment of fault hypotheses against real systems, in the lab and in the field. Model checkers and test systems complement each other. So do, for instance, SALT and ALFHA.

When fault-injection tests shall be applied to AUTOSAR [2] systems, an obvious idea is to use tracing and debugging modules belonging to the basic software (BSW). However, these BSW modules do not enable cycle-accurate data seeding because the specifications lack necessary provisions. Additionally, the debugger specification does not define the linkage to the operating system and hardware. Despite of a comprehensive meta-model for automatic software system configuration, AUTOSAR has no notion of a real-time database cross cutting the AUTOSAR's software component layer (SWC), the AUTOSAR run-time environment (RTE) and the BSW of one or more electronic control units (ECUs). In Summary, AUTOSAR does not fulfill the requirements on target systems as specified in Section II. Similar arguments hold for the Software-Implemented Fault Injection (SWIFI) technique presented by Pintard et al. [10] for two reasons: (1) SWIFI targets AUTOSAR, (2) SWIFI is intrusive because it modifies source code. The framework for instrumentation of AUTOSAR systems from Piper et al. [11] operates at model level and is intrusive because it changes source or binary code of purposefully selected system parts. *Routine specialization* [4] is another intrusive SWIFI technique although it reduces the impact on the system per type of fault injected.

Automatic checks of fault hypotheses demand deep system understanding, because ALFHA tests access system internals represented by signals and data in one or more real-time system databases of distributed systems. Kane et al. [6] present an approach for formulating tests that explore and check the behavior of distributed systems solely on the network level where an external run-time monitor performs offline verification. As the authors state, their approach does well support offline tests but pose challenges when used online because of the possible disturbance of the behavior of real-time systems. Another issue is the limited observability and controllability of systems under test.

VIII. Summary and Future Work

Safety properties must be proven to hold in case of faults occurring for an integrated system for it to be called safe with confidence. In practice, tests that can falsify truth statements are the method of choice for proving fault-hypotheses of complex, safety-critical systems. For this purpose the domain-specific language ALFHA was designed. ALFHA tests allow reasoning about time-constrained properties of distributed, redundant real-time systems in typical, critical and exceptional situations. Time-bounded runs of target systems prove assertions of system properties in ALFHA being either true or false.

Systems which can be evaluated for the validity of ALFHA statements have many advantages. ALFHA tests observe and trigger a target system at several locations in space and time without distorting its behavior. Systems designed for ALFHA tests execute module operations in a time-triggered manner providing deterministic behavior. Data flows between and within modules are stored cyclically in a database on each system node. The database is the key means for decoupling system modules and the starting point for non-intrusive data-seeding and data capturing. In future work, more tests of different types and with different goals are planned in order to test and elaborate the scope of ALFHA and ALFHA traces. Compiling ALFHA tests automatically to VITE tests is a promising approach. ALFHA tests shall be extended with automatic plausibility analysis, test coverage determination, and offer suggestions for isolating or replacing parts of the system that jeopardize the reliance of safety-critical systems.

References

[1] ISO/DIS 26262 Road vehicles – Functional safety – Part 1-10. Status: Published, November 2011.

[2] AUTomotive Open System ARchitecture (AUTOSAR) Release 4.2. Technical report, October 2014.

[3] A. Bauer, M. Leucker, and J. Streit. SALT - Structured Assertion Language for Temporal Logic. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 757–775, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] J. Cunha, M. Rela, and J. Silva. Can Software Implemented Fault-Injection be Used on Real-Time Systems? In J. Hlavika, E. Maehle, and A. Pataricza, editors, *Dependable Computing EDCC-3*, volume 1667 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999.

[5] J. Fröhlich and R. Schmid. Architecture for a Hard-Real-Time System Enabling Non-intrusive Tests. In *25th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, page 24, November 2014.

[6] A. Kane, T. Fuhrman, and P. Koopman. Monitor Based Oracles for Cyber-Physical System Testing: Practical Experience Report. In *Dependable Systems and Networks (DSN), 44th Annual IEEE/IFIP International Conference on*, pages 148–155, June 2014.

[7] H. Kopetz. On the Fault Hypothesis for a Safety-Critical Real-Time System. In M. Broy, I. Krger, and M. Meisinger, editors, *Automotive Software Connected Services in Mobile Networks*, volume 4147 of *Lecture Notes in Computer Science*, pages 31–42. Springer Berlin Heidelberg, 2006.

[8] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[9] R. Obermaisser and P. Peti. A Fault Hypothesis for Integrated Architectures. In *Intelligent Solutions in Embedded Systems, 2006 International Workshop on*, pages 1–18, June 2006.

[10] L. Pintard, M. Leeman, A. Ymlahi-Ouazzani, J. Fabre, et al. Using Fault Injection to Verify an AUTOSAR Application According to the ISO 26262. *SAE Technical Paper*, January 2015.

[11] T. Piper, S. Winter, P. Manns, and N. Suri. Instrumenting autosar for dependability assessment: A guidance framework. In *Dependable Systems and Networks (DSN), 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2012.

[12] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll. 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.