

# Specification of Conditions for Error Diagnostics

Christof Efkemann<sup>1,2</sup>

*TZI  
University of Bremen  
Bremen, Germany*

Tobias Hartmann<sup>1,3</sup>

*TZI  
University of Bremen  
Bremen, Germany*

---

## Abstract

This paper describes the basic concepts of error diagnostics and an associated rule system whose application helps to identify potential hardware/software locations of errors which caused a failure observed while executing tests of embedded systems. Further on, an overview over the “Avionics Smoke Detection System” is given, where the main algorithms have been applied. The methods, techniques and tools described here rely on the preceding investigations performed with respect to signal flow and the observation of causal chains in distributed test benches during test suite executions. These results have been elaborated in the KATO project within the German aerospace research programme LUFO III.

*Keywords:* Embedded systems, error diagnostics, RT-Tester.

---

## 1 Introduction

This document describes the results achieved by the University of Bremen, TZI, for KATO-TP13 work packages “*Specification of conditions for error diagnostics*” and “*Implementation of an error diagnostic prototype*”. With almost ten years of experience in the area of test and verification of avionics controllers all kinds of faults, errors and anomalies have been observed and detected by our workgroup. The obvious next step is to classify the encountered faults. Based on the fault models commonly used in semiconductor fault diagnostics[1][6] we applied these models and methods and provide a complete strategy for locating faults in distributed embedded

---

<sup>1</sup> The authors have been partially supported by the BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015B

<sup>2</sup> Email: [chref@tzi.de](mailto:chref@tzi.de)

<sup>3</sup> Email: [hartmann@tzi.de](mailto:hartmann@tzi.de)

systems within their environments. The methods, techniques and tools illustrated here rely on preceding investigations, which have been documented in [7].

Due to legal constraints we cannot present “real” source code, but instead we illustrate our methods for error diagnostics with a slightly simplified and abstracted example of a real system. The System Under Test (SUT) and its testing environment are depicted in Fig. 1 and introduced in the paragraphs below.

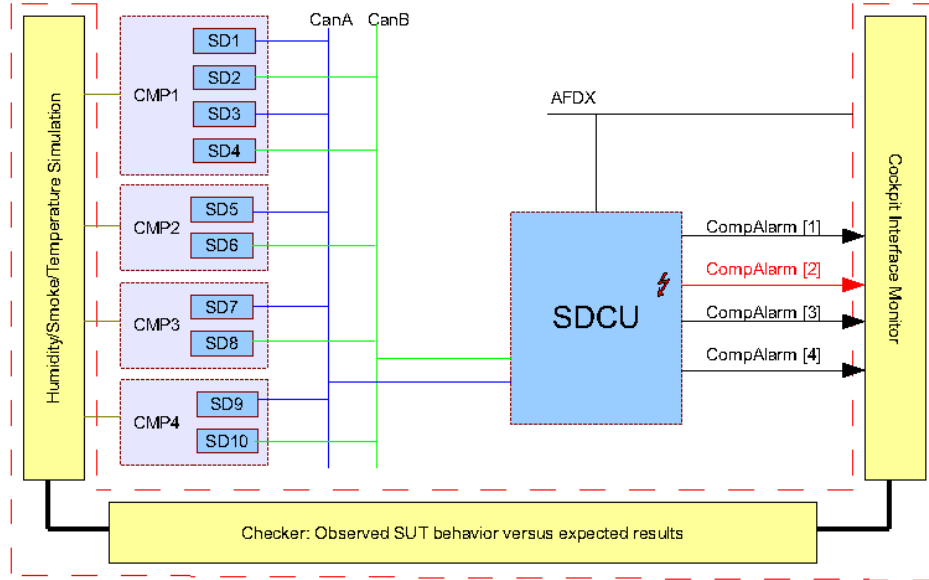


Fig. 1. Testing scenario

A simplified version of an aircraft *smoke detection system* is used as the SUT. Its model, which is assumed to be correct, and the corresponding faulty implementation are described in Section 2. The central control unit (*smoke/fire detection control unit* (SDCU)) contains an error similar to one of those errors found during “real” controller testing.

The SDCU is connected to the cockpit via a bus system called Avionics Full Duplex Switched Ethernet (AFDX), which is commonly used in modern aircraft. This bus is used to send the alarm states of the compartments to the cockpit. These messages are pictured as arrows, leading from the SDCU to the cockpit.

However, to emphasize the fault, the reporting message is made visible in the test scenario overview. The faulty part is marked by a red arrow.

To keep the overview short and simple, the flow of the messages from the SDCU to the sensors - and vice versa - are not shown. They are routed within the two CAN buses A and B.

The implementation is embedded into a testing and simulation environment, using *Verified’s* RT-Tester tool[12]. The resulting testing environment and the test suite executed therein are described in Section 3. In Section 4, the application of the diagnostic tool kit described in the preceding chapters is illustrated for the purpose of identifying the causes and locations of the test failures observed.

## 2 SDF SUT Model and Implementation

The system consists of (see Fig. 1 above) the following items:

- The **smoke detectors (SD)** measure humidity, smoke and temperature at designated locations in the aircraft and transmit smoke/fire alarms and other status information on a CAN bus system.
- The **smoke detection control unit (SDCU)** collects sensor data from various busses and distributes alarms and other messages to the cockpit (and other systems).
- The **communication media** consist of
  - CAN busses for SD ↔ SDCU communication,
  - AFDX bus for SDCU ↔ Cockpit communication,
  - discrete I/O lines for cockpit indications.

Observe that real smoke detection systems use several additional communication interfaces, in particular, aural and visual indication to the aircraft cabin. For the purpose of illustrating our concepts for error diagnostics, however, these interfaces and their associated additional functionality are not relevant. The central SUT component is the SDCU controller. One of the most important SDCU methods is `processMessages()`. Here, the decisions whether to raise compartment smoke alarms are made. In the correctly modelled system the alarm status of compartment  $c$  depends on the conjunction of local alarm states. For compartment Comp1 the states of the sensors SD1 to SD4 are evaluated. Two sensors are used for the compartments Comp2 to Comp4 respectively. This leads to the following alarm equations:

- Comp1:

$$\begin{aligned} \text{compAlarm}[1] = & (\text{sensorSDSAlarm}[1] \wedge \text{sensorSDSAlarm}[2]) \vee \\ & (\text{sensorSDSAlarm}[3] \wedge \text{sensorSDSAlarm}[4]) \end{aligned}$$

- Comp2 ... Comp4:

$$\begin{aligned} \text{compAlarm}[c] = & \text{sensorSDSAlarm}[2c + 1] \wedge \text{sensorSDSAlarm}[2c + 2], \\ c \in & \{2 \dots 4\} \end{aligned}$$

However, the SUT implementation contains an error: For compartment Comp2, the alarm decision depends on  $\text{sensorSDSAlarm}[5] \wedge \text{sensorSDSAlarm}[13]$  instead of  $\text{sensorSDSAlarm}[5] \wedge \text{sensorSDSAlarm}[6]$ . In the diagnostic procedure described below, this failure will be referred to as *Failure*.

## 3 SDF Testing Environment

In this document we focus on *system integration testing*, so the tests will typically be

- **Black-box tests** on controller level: Since controllers have been tested by their suppliers during the mandatory HW/SW integration test suite, the system tests will deal with controllers as black-boxes.
- **Black-box tests** on peripheral device level (same reason as for controllers).

- **Grey-box tests** with respect to inter-controller communications: During system integration testing, network monitors are typically available to record at least a portion of the data exchange between communicating controllers. For example, AFDX monitors and ARINC 429 monitors can be used to record snapshots or specific types of data packages. Some communications, however, often cannot be observed, such as, for example, the data exchange between redundant fault-tolerant controllers.
- **Grey-box tests** with respect to controller  $\leftrightarrow$  device communication: Just like inter-controller communications, the data exchange between controllers and their directly connected peripherals can be – at least partially – observed using bus monitors (e.g. monitors for CAN busses, ARINC 429 busses), or measurement equipment (e.g. for discrete and analogue I/O interfaces).

The RT-Tester tool can be used for all kinds of tests – from unit level tests to hardware/software integration tests (described in [5]). However, in this document we focus on system integration tests, and all mentioned tests were executed using this tool.

## 4 Diagnostic Procedure

### 4.1 Step 1: Initial Black-Box Test Results

A diagnostic procedure is always triggered by a discrepancy observed during a test suite. The failure – contained in the SDCU itself – is detected during a functional test of compartment smoke alarms in compartment Comp2.

The initial values for the compartments are shown in the first test log. The values for humidity and smoke are recorded as percent values. The temperature is given in degrees Celsius.

```
TM 00027023005 AM 4 0 : ( 25 ) ENVIRONMENT:
: -----
: COMP: 1 2 3 4
: HUMIDITY: 5 5 5 5
: SMOKE: 2 2 2 2
: TEMP: 20 20 20 20
```

The test environment simulates a relevant smoke and slight temperature increase for Comp2, which is recorded in the test execution protocol as

```
TM 00029035001 AM 4 0 : ( 27 ) ENVIRONMENT:
: -----
: COMP: 1 2 3 4
: HUMIDITY: 5 5 5 5
: SMOKE: 90 90 90 2
: TEMP: 30 30 30 20
```

The threshold values for alarms are at 80% for smoke and 65° for temperature. Therefore an alarm must be reported for three different compartments: Comp1, Comp2 and Comp3. However, no alarm is reported for the compartment Comp2:

```

TM 00034035267 AM   4 0 : ( 32 ) SDCU:
                        : -----
                        :      COMP: 1  2  3  4
                        :      ALARM: 1  0  1  0

```

This test step shows that after 5 seconds (timestamp unit: microseconds), there is still no smoke alarm for compartment Comp2, leading to test failure.

#### 4.2 Step 2: Interface Analysis – Generation of the Causal Graph – Version 1

The causal graph shows all components and interfaces that may possibly influence the values passed along interface `compAlarm[2]`. For the version 1 representation of the graph all components which are *physically* connected are considered, because bridging faults<sup>4</sup> in components may result in arbitrary data flows, as long as a physical connection is available. As soon as certain types of internal component faults can be excluded, the causal graph may be narrowed, leading to new versions. Conversely, a more detailed analysis of a potentially faulty component may require to analyse its *internal data flow*. This leads to a *refinement* of the causal graph version  $n$ , where a component node  $C$  in version  $n$  is replaced by a detailed data flow network representing communications within  $C$  and leading to a new version  $n+1$  of the graph. The initial version of the causal graph for the failure detected on interface `compAlarm[2]` is shown in Fig. 2. The graph contains cycles, because messages are transported from smoke/fire detectors to the SDCU and vice versa, and the messages from SDCU to SDs can also influence the state on interface `compAlarm[2]`: For example, if the SDCU would fail to poll sensors SD5 and SD6, then no alarm messages would be sent from SD5 and SD6 to the SDCU.

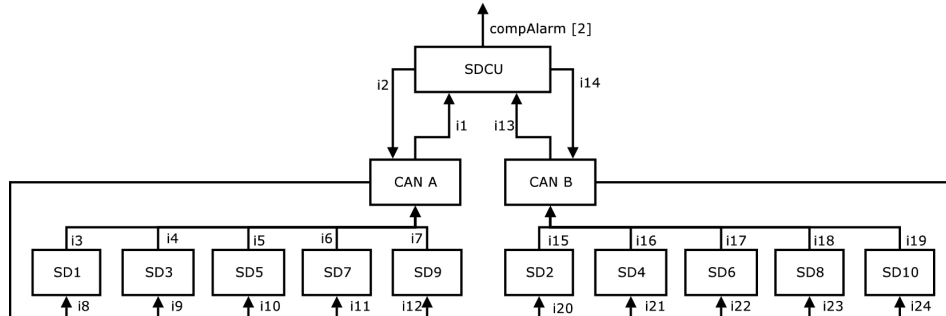


Fig. 2. Causal graph, version 1, for Failure

#### 4.3 Step 3: Interface Analysis – Generation of the Fault Tree

The fault tree[8][11, pp. 43ff] constructed in Step 3 depicts the possible error hypotheses, together with the boundary conditions which must hold in order to make a hypothetical error cause the observed failure on the interface between SUT and testing environment. The error classification used for each component follows the fault models introduced in [7, pp. 13ff] and the fault tree construction technique described in [7, pp. 6ff]. In the diagnostic procedure described here we omit the

<sup>4</sup> For integrated circuits this is discussed in [10, pp. 335ff]

possibility of an external intruder (see [7]) because we are dealing with a closed system whose components are well-known.

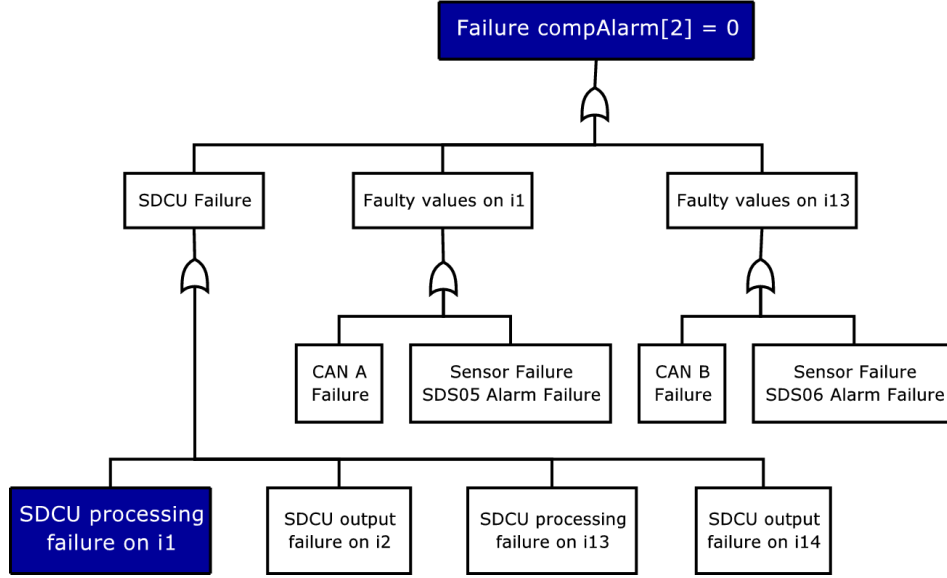


Fig. 3. Fault tree associated with interface analysis for Failure

Following the paths in the causal graph of Fig 2, we can immediately derive the first levels of the fault tree, as depicted in Fig. 3: The failure on output interface `compAlarm[2]` might be directly caused by an internal SDCU failure or by an erroneous input on the interfaces `i1`, `i13` between SDCU and the CAN bus. SDCU-internal failures may either cause faulty transformations from `i1`- and `i13`-inputs to `compAlarm[2]` outputs or by output faults on interfaces `i2` and `i14`, which may in turn cause unexpected smoke detector behaviour, so that the alarm messages of the crucial detectors SD5, SD6 are either not sent at all or not accepted by the SDCU. Faulty input values to a correctly operating SDCU may either be caused by a failure of the CAN bus or by sensor failures. Since compartment alarms indicated on output `compAlarm[c]`,  $c \in \{1 \dots 4\}$  require that both<sup>5</sup> associated sensors `SD(2c+1)`, `SD(2c+2)` signal an alarm, the crucial sensors for the observed failure on `compAlarm[2]` are SD5 and SD6.

Refining the possibilities of SDCU processing failures on interface `i1` and `i13` leads to fault sub-tree 4: Not showing fault-types which are *a priori* highly improbable leaves us with 5 potential types of fault (as classified in [3, chap. 7]):

- (i) **Bridging faults** would arise if – due to internal interface errors of the SDCU – the `compAlarm[2]` output could be influenced by the values of other inputs which should be disregarded in a correctly operating SUT: If the SDCU used the wrong sensors (SD1,2,3,4,7,8,9,10) to determine the output on `compAlarm[2]` and these sensors were not in alarm state, then no compartment alarm would be raised on `compAlarm[2]`.
- (ii) **Signal deletion errors** would arise if the `i1` or `i13` input would not be relayed internally to the SDCU sub-component responsible for raising the

<sup>5</sup> Comp1 is an exception: there are four sensors installed

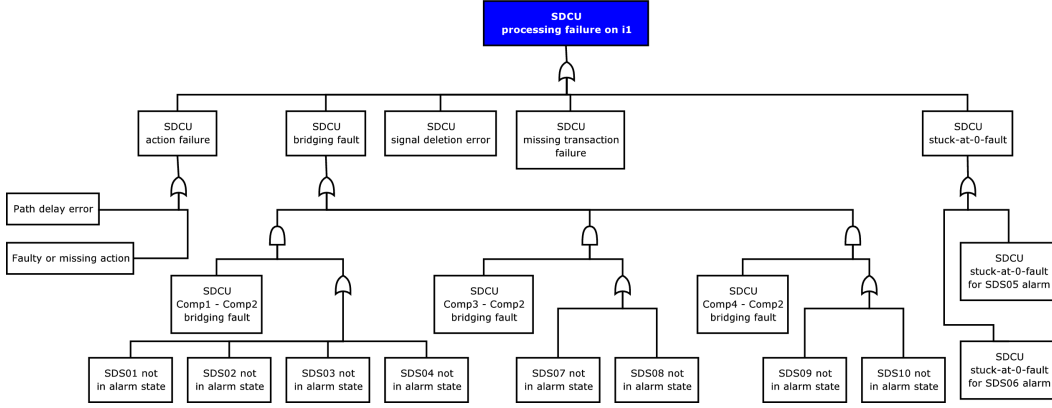


Fig. 4. Fault sub-tree associated with potential SDCU processing failures on interface  $i1$  which might cause Failure

`compAlarm[2]` compartment alarm.

- (iii) **Stuck-at-0 faults** occur if the guard condition for raising `compAlarm[2]` to 1 were faulty, so that it always evaluates to *false*.
- (iv) An **action failure** would arise if the assignment of the new value 1 to `compAlarm[2]` were faulty.
- (v) A **missing transition** failure would arise if inputs from sensors SD5 and SD6 on interface  $i1$ ,  $i13$  would simply not trigger any action at all.

Observe that the fault tree can be set up incrementally, so that extensions of leaves are only performed after other branches of the tree have been pursued which did not help to uncover the failure under investigation. We will therefore analyse the potential SDCU failures further, before refining the other branches of the initial fault tree 3.

#### 4.4 Step 4: Falsification of Fault Hypotheses

After the fault tree associated with the interface analysis has been elaborated, the fault hypotheses implied by nodes of the tree can be falsified one by one, until the cause for the observed failure has been identified. Falsification can be performed by

- (i) analysing additional data recorded during the test,
- (ii) performing additional system integration tests designed to verify/falsify a specific fault hypothesis,
- (iii) performing additional lower-level tests (e. g. HW/SW integration tests or unit tests), in order to observe additional interfaces which were not visible on system integration test level,
- (iv) code execution or interpretation using concrete values,
- (v) abstract code interpretation using interval values (like described in [2]),
- (vi) informal or formal code analysis (inspection, formal verification etc.).

Obviously, method 1 is the most preferable one since it neither requires additional tests nor source code availability. Following the fault model from [7], it will now be analysed how specific tests can be used to falsify fault hypotheses associated

with a given type of error.

#### 4.4.1 Falsification tests for bridging faults.

Bridging faults imply that unintended inputs  $x_i, i = 0, 1, 2, \dots$  to the erroneous component influence a certain output  $y$  in an illegal way. A falsification test therefore consists of systematically changing the inputs  $x_i$  and observing whether a certain combination can illegally stimulate output changes on  $y$ . For the potential bridging faults of the SDCU as depicted in Fig. 4 the interpretation is as follows, as can be seen from the fault tree:

- If the observed error is due to a bridging fault from Comp1 sensor values to Comp2 sensor values, then it can only arise if sensors SD1,2,3,4 are *not* in ALARM state: Otherwise we would observe a correct output `compAlarm[2]=1`, though illegally caused by the wrong sensors SD1,2,3,4.
- Similarly, if the observed error is due to a bridging fault from Comp3 sensor values to Comp2 sensor values, then it can only arise if sensors SD7,8 are *not* in ALARM state.
- Analogously, if the observed error is due to a bridging fault from Comp4 sensor values to Comp2 sensor values, then it can only arise if sensors SD9,10 are *not* in ALARM state.

Analysing the test execution log, we see that when the output failure `compAlarm[2]=0` is observed, sensors SD1,2,3,4 are already in state ALARM, and the Comp1 alarm has been indicated correctly. As a consequence, there can be no bridging fault from Comp1 sensor values to Comp2 sensor values. Also, the sensors SD6,7 are in state ALARM and the compartment alarm for Comp3 is displayed: there can be no bridging fault from Comp3 to Comp2 either. The simulated environment for all compartments show that the thresholds for smoke are exceeded in the compartments Comp1, Comp2 and Comp3 (see log at timestamp 00029035001  $\mu s$ , p. 4). The log at timestamp 00034035267  $\mu s$  (see p. 5) shows that no alarm was detected for Comp2.

```

TM 00029024572 AM   4 0 : ( 27 ) SENSORS SDS:
: -----
: SENSOR ID: 1  2  3  4  5  6  7  8  9 10
:   ALARM:  1  1  1  1  1  1  1  1  0  0
:   DEFECT:  0  0  0  0  0  0  0  0  0  0
:   FAILURE: 0  0  0  0  0  0  0  0  0  0
:   STANDBY: 0  0  0  0  0  0  0  0  1  1

```

Unfortunately, the sensors SD9,10 were still in state STANDBY when the failure occurred, so the test run could not exclude the bridging fault from Comp4 sensor values to Comp2 sensor values: We need an additional test where the sensors SD9,10 are permanently in state ALARM. If this does not lead to an output `compAlarm[2]=1`, this also excludes the second potential bridging fault.

**Test: Bridging**

In this test, the environment for compartment Comp4 is set in a way that the installed sensors will report an alarm. This can be read from the execution log at timestamp 00029028720  $\mu s$ .

```

TM 00029028720 AM   4 0 : ( 27 ) ENVIRONMENT:
                        : -----
                        :   COMP:   1   2   3   4
                        : HUMIDITY:   5   5   5   5
                        :   SMOKE:  90  90  90  90
                        :   TEMP:   30  30  30  30

```

The Sensors SD9,10 are set to state alarm, which is reported at timestamp 00029029193  $\mu s$ .

```

TM 00029029193 AM   4 0 : ( 27 ) SENSORS SDS:
                        : -----
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :   ALARM:  1  1  1  1  1  1  1  1  1  1
                        :   DEFECT:  0  0  0  0  0  0  0  0  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  0  0

```

As expected, the SDCU sets the alarm for compartment Comp4.

```

TM 00029029894 AM   4 0 : ( 27 ) SDCU:
                        : -----
                        :   COMP:  1  2  3  4
                        :   ALARM:  1  0  1  1

```

As shown in this test, the occurrence of a bridging fault can be excluded.

*4.4.2 Falsification tests for signal deletion errors.*

Systematic tests for falsification of signal deletion errors can be designed and executed if the SUT has the following proven property:

If signal value  $v_0$  is correctly transmitted over interface  $i$ , then value  $v_1$  must be correctly transmitted, too.

Such a hypothesis is fulfilled<sup>6</sup>, for example, if  $i$  is a shared variable interface and all messages placed into  $i$  have the same length and a checksum, allowing to detect corrupted (in particular, truncated) messages.

If the hypothesis holds, an additional test stimulating  $v_0$  can be executed. If it passed then this implies that  $v_1$  was not lost within the SUT. In our case, the hypothesis is fulfilled by the SUT implementation. Therefore an additional test can be created where another state value of sensors SD5,6 is stimulated, for instance with a FAILURE state. When the sensors transmit the state *failed*, the SDCU is expected to mark the sensors as failed. If this occurs, we have proven that sensor

<sup>6</sup> It is not fulfilled, for example, if signal value  $v_0$  is a low bit in a 32-bit command word  $w$  and  $v_1$  is a high bit in  $w$ , so that, after illegally masking the higher bits of  $w$ ,  $v_0$  is still correctly visible, but  $v_1$  has been illegally set to 0.

state changes of SD5,6 are not lost within the SDCU. Since the SUT implementation really shows the expected messages, a signal deletion error can be excluded.

### Test: Signal-deletion

In this test, the sensors SD5,6 constantly send the state failure to the SDCU. The internal state of the sensors is not modified, just the value in the request frame is adjusted. As it is recorded at TM 00029031597  $\mu s$ , the sensors are marked failed. Therefore the occurrence of a signal deletion fault can be excluded.

```
TM 00029031597 AM 4 0 : ( 27 ) SDCU:
: -----
: SENSOR ID: 1 2 3 4 5 6 7 8 9 10
: ALARM: 1 1 1 1 0 0 1 1 0 0
: FAILURE: 0 0 0 0 1 1 0 0 0 0
: STANDBY: 0 0 0 0 0 0 0 0 1 1
: COMP: 1 2 3 4
: ALARM: 1 0 1 0
```

#### 4.4.3 Falsification tests for path delay errors.

A path delay error occurs when the specified transition is not completely lost, but occurs too late and/or needs an additional stimulus to be triggered. This type of error would be revealed by waiting for a sufficiently long time for the expected output to occur (of course, an upper bound must be known to perform this test), and toggling the inputs in a way that should stimulate *a sequence of output value changes* instead of only one change. In our case, the additional test would consist of exercising an input sequence on the Comp2 temperature and smoke status which would result in a sequence of ALARM  $\rightarrow$  STANDBY  $\rightarrow$  ALARM  $\rightarrow$  STANDBY  $\rightarrow$  ... state changes of sensors SD5,6.

### Test: Path-delay

In this test, the environment for compartment Comp2 is modified in a way that the sensors will send a sequence of ALARM  $\rightarrow$  STANDBY  $\rightarrow$  ALARM  $\rightarrow$  STANDBY  $\rightarrow$  ... to the SDCU. As can be seen from the test log, the environment alternates the values for smoke and temperature and the corresponding sensors change their state accordingly. The environment is set to “no fire”, so are the sensors SD5,6:

```
TM 00010026978 AM 4 0 : ( 18 ) ENVIRONMENT:
: -----
: COMP: 1 2 3 4
: HUMIDITY: 5 5 5 5
: SMOKE: 90 2 90 2
: TEMP: 30 20 30 20

TM 00010027377 AM 4 0 : ( 18 ) SENSORS SDS:
: -----
: SENSOR ID: 1 2 3 4 5 6 7 8 9 10
```

```

:     ALARM: 1  1  1  1  0  0  1  1  0  0
:     DEFECT: 0  0  0  0  0  0  0  0  0  0
:     FAILURE: 0  0  0  0  0  0  0  0  0  0
:     STANDBY: 0  0  0  0  1  1  0  0  1  1

```

```

TM 00010027856 AM  4 0 : ( 18 ) SDCU:
: -----
:     COMP: 1  2  3  4
:     ALARM: 1  0  1  0

```

In the next test step, the environment is set to “fire” and the sensors change their states accordingly:

```

TM 00010526998 AM  4 0 : ( 19 ) ENVIRONMENT:
: -----
:     COMP:   1   2   3   4
: HUMIDITY:  5   5   5   5
:     SMOKE: 90  90  90   2
:     TEMP:  30  30  30  20

```

```

TM 00010527398 AM  4 0 : ( 19 ) SENSORS SDS:
: -----
: SENSOR ID: 1  2  3  4  5  6  7  8  9 10
:     ALARM: 1  1  1  1  1  1  1  1  0  0
:     DEFECT: 0  0  0  0  0  0  0  0  0  0
:     FAILURE: 0  0  0  0  0  0  0  0  0  0
:     STANDBY: 0  0  0  0  0  0  0  0  1  1

```

```

TM 00010527908 AM  4 0 : ( 19 ) SDCU:
: -----
:     COMP: 1  2  3  4
:     ALARM: 1  0  1  0

```

Again, the environment is set to “no alarm”. The sensors are set to state standby.

```

TM 00011027004 AM  4 0 : ( 20 ) ENVIRONMENT:
: -----
:     COMP:   1   2   3   4
: HUMIDITY:  5   5   5   5
:     SMOKE: 90   2  90   2
:     TEMP:  30  20  30  20

```

```

TM 00011027416 AM  4 0 : ( 20 ) SENSORS SDS:
: -----
: SENSOR ID: 1  2  3  4  5  6  7  8  9 10
:     ALARM: 1  1  1  1  0  0  1  1  0  0
:     DEFECT: 0  0  0  0  0  0  0  0  0  0
:     FAILURE: 0  0  0  0  0  0  0  0  0  0
:     STANDBY: 0  0  0  0  1  1  0  0  1  1

```

```

TM 00011027874 AM 4 0 : ( 20 ) SDCU:
: -----
:      COMP: 1  2  3  4
:      ALARM: 1  0  1  0

```

Also it can be noticed that the SDCU is not changing the state for Comp2. Therefore the occurrence of a path delay error can be excluded.

#### 4.4.4 Falsification tests for missing transition errors.

An omitted transition means that the SUT illegally remains in a given state, though a new trigger event occurred. This situation can be uncovered by black-box tests, using the *characterisation traces* of the state machine defining the transition (see [4] for details): The test execution is extended by additional input sequences, so that the associated SUT reactions reveal whether the required transition has been performed or not.

For our system, a command to dump the internal BITE (Built-In Test Equipment) memory can be used as such an additional input sequence<sup>7</sup>: Every alarm is not only indicated on the `compAlarm[c]` interfaces, but also recorded in the internal BITE memory whose contents can be accessed using commands sent to the SDCU via AFDX.

#### 4.4.5 Falsification tests for Stuck-at-0/1 errors.

For suspected stuck-at-0/1 errors<sup>8</sup>, a test is designed in such a way that a change in the output interface  $y$  where the failure was observed only depends on a single input  $x$ . Then  $x$  is changed in such way that it should trigger corresponding  $y$  changes. If these occur, a “stuck-at situation” cannot be present.

### Test: Stuck-at-zero

In our case, we perform a test where SD5 stays continuously in state ALARM, while the state of SD6 is toggled between STANDBY and ALARM. Then, in a correctly operating SUT, an output `compAlarm[2]=1` should occur if and only if SD6 is in state ALARM. In the following test log, it can be seen that no compartment alarm for Comp2 is raised because only sensor SD5 is in state ALARM whereas sensor SD6 remains in state STANDBY. This is the expected behaviour.

```

TM 00021029341 AM 4 0 : ( 20 ) SDCU:
: -----
:  SENSOR ID: 1  2  3  4  5  6  7  8  9 10
:      ALARM: 1  1  1  1  1  0  1  1  0  0
:      FAILURE: 0  0  0  0  0  0  0  0  0  0
:      STANDBY: 0  0  0  0  0  1  0  0  1  1
:      COMP: 1  2  3  4
:      ALARM: 1  0  1  0

```

<sup>7</sup> Not shown here due to space constraints.

<sup>8</sup> For integrated circuits this is discussed in [10, pp. 335ff]

```

TM 00022029307 AM 4 0 : ( 21 ) SDCU:
: -----
: SENSOR ID: 1 2 3 4 5 6 7 8 9 10
: ALARM: 1 1 1 1 1 0 1 1 0 0
: FAILURE: 0 0 0 0 0 0 0 0 0 0
: STANDBY: 0 0 0 0 0 1 0 0 1 1
: COMP: 1 2 3 4
: ALARM: 1 0 1 0

```

In the next steps, both sensors are in state ALARM, which can be seen in the test execution log. But still no alarm for compartment Comp2 is signalled.

```

TM 00023029318 AM 4 0 : ( 22 ) SDCU:
: -----
: SENSOR ID: 1 2 3 4 5 6 7 8 9 10
: ALARM: 1 1 1 1 1 1 1 1 0 0
: FAILURE: 0 0 0 0 0 0 0 0 0 0
: STANDBY: 0 0 0 0 0 0 0 0 1 1
: COMP: 1 2 3 4
: ALARM: 1 0 1 0

```

```

TM 00024029314 AM 4 0 : ( 23 ) SDCU:
: -----
: SENSOR ID: 1 2 3 4 5 6 7 8 9 10
: ALARM: 1 1 1 1 1 1 1 1 0 0
: FAILURE: 0 0 0 0 0 0 0 0 0 0
: STANDBY: 0 0 0 0 0 0 0 0 1 1
: COMP: 1 2 3 4
: ALARM: 1 0 1 0

```

As a consequence, this test could not falsify the stuck-at-0 fault.

#### 4.4.6 Repeated Application of Steps 3 and 4.

If all fault hypotheses in the fault tree elaborated in Step 3 have been falsified, but the tree has not been completely refined, then we return to Step 3 to further extend selected leaves of the fault-tree. After that, Step 4 is repeated.

If the tree was completely refined and no error was found during the execution of these steps, there is a good chance that one of the additional tests failed to cover the problem. In this case there are only two ways of detecting the error: either to start at the end of the trees and do “backtracking” or to start at the beginning, performing validations for each additional test. The choice of further actions then depends on the specific problem and will not be elaborated here.

#### 4.5 Step 5: Error Identification

In our case, all fault hypotheses but the potential stuck-at-0 fault can be falsified by means of additional tests. The internal structure of the SDCU must be taken into account, so that the global input variables `sensorSDSAlarm[]` of the

`processMessages()` method become visible. An additional unit test of this method reveals the presence of a stuck-at-0 fault within this method.

In order to aid with error diagnosis on source-code level a software tool has been developed: Using the interval analysis techniques described in [7, pp. 9ff] (based on [9]), the Interactive Interval Analyser can easily help to identify the error. The Interval Analyser works on compiler-generated control flow graph (CFG) information. The user can select a CFG (which corresponds to a function) from the compilation unit. Subsequently a window containing the function’s inputs and outputs is displayed: The inputs are global variables as well as function parameters, while the outputs are again global variables and the function’s return value.

The user may now change the intervals assigned to the inputs and the Analyser will interactively show the impact of the changes on the outputs (Fig. 5).

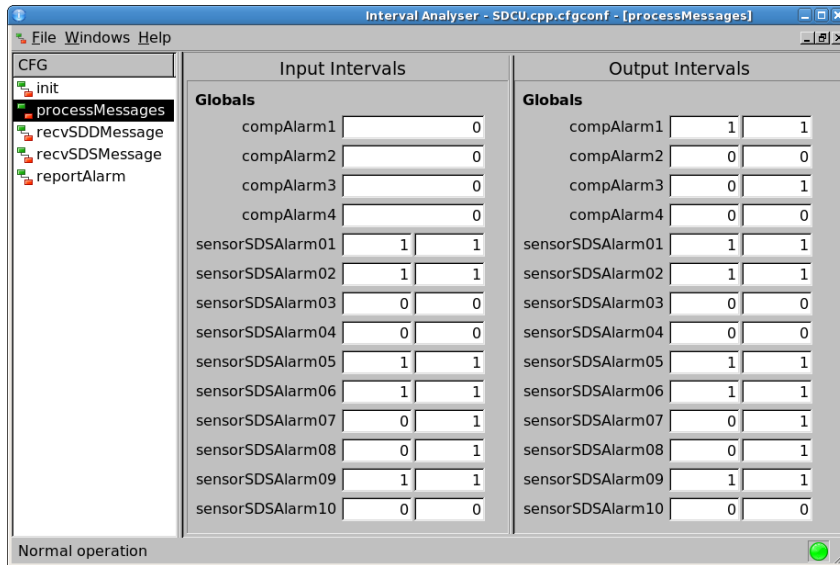


Fig. 5. Interval Analyser

The Analyser can also be used in combination with Verified’s *RT-Tester User Interface* (RTTUI). It is then possible to observe the code coverage achieved with the current input intervals. The RTTUI will display the CFG file<sup>9</sup> and colourise all blocks that have been reached during the last execution. This feature can be used for error diagnosis: Setting both inputs `sensorSDSAlarm[5]` and `sensorSDSAlarm[6]` to the interval `[1,1]` does not lead to coverage of block 8 (where `compAlarm[2]` would have been set to 1), as shown in Fig. 6.

The last step is to match the corresponding lines of code to the block 8 of the CFG. This leads to the position where the error is located within the source code:

```
if ( sensorSDSAlarm[5] && sensorSDSAlarm[13] ) {
    compAlarm[2] = 1;
}
```

This reveals the observed error: for compartment `Comp2`, the alarm decision depends on `sensorSDSAlarm[5]  $\wedge$  sensorSDSAlarm[13]` instead of

<sup>9</sup> Displaying the original C/C++ source code is currently not possible. This feature is planned for a later version.

$sensorSDSAlarm[5] \wedge sensorSDSAlarm[6]$ .

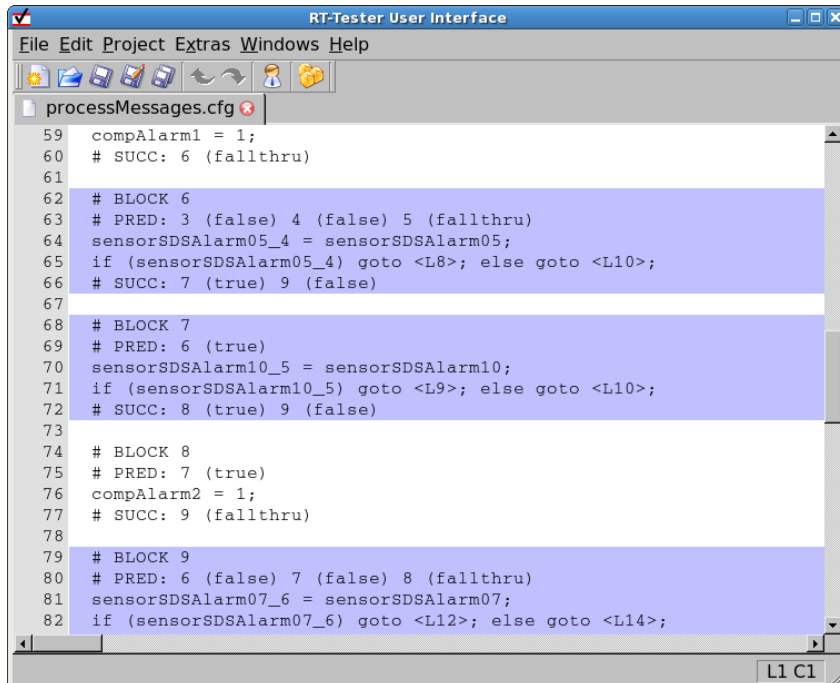
## 5 Conclusion

Our experience has shown that it is possible to locate any kind of fault when following the presented procedures (and refinements). Being state of the art for semiconductor error diagnostics, to our knowledge the presented strategy for fault detection is not yet commonly used within the area of distributed embedded systems. Through the tight integration of the procedures with the tool chain a reliable and efficient way of testing is introduced. The tool chain, consisting of the RT-Tester, its user interface (RTTUI) and the Interval Analyser, can be supplemented by tools<sup>10</sup> like [Relex FTA](#) or [Isograph FaultTree+](#) for fault tree analysis.

## References

- [1] J. A. Abraham and W. K. Fuchs. Fault and error models for VLSI. *IEEE Proceedings*, 74:639–654, May 1986.
- [2] Bahareh Badban, Martin Fränze, Jan Peleska, and Tino Teige. Test automation for hybrid systems. 2006. Submitted to Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006), Available under [http://www.informatik.uni-bremen.de/agbs/projects/hybris/peleska\\_et\\_al\\_soqua2006.pdf](http://www.informatik.uni-bremen.de/agbs/projects/hybris/peleska_et_al_soqua2006.pdf).
- [3] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [4] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.

<sup>10</sup>No tool support was used for the fault trees in this paper.



```

RT-Tester User Interface
File Edit Project Extras Windows Help
processMessages.cfg
59 compAlarm1 = 1;
60 # SUCC: 6 (fallthru)
61
62 # BLOCK 6
63 # PRED: 3 (false) 4 (false) 5 (fallthru)
64 sensorSDSAlarm05_4 = sensorSDSAlarm05;
65 if (sensorSDSAlarm05_4) goto <L8>; else goto <L10>;
66 # SUCC: 7 (true) 9 (false)
67
68 # BLOCK 7
69 # PRED: 6 (true)
70 sensorSDSAlarm10_5 = sensorSDSAlarm10;
71 if (sensorSDSAlarm10_5) goto <L9>; else goto <L10>;
72 # SUCC: 8 (true) 9 (false)
73
74 # BLOCK 8
75 # PRED: 7 (true)
76 compAlarm2 = 1;
77 # SUCC: 9 (fallthru)
78
79 # BLOCK 9
80 # PRED: 6 (false) 7 (false) 8 (fallthru)
81 sensorSDSAlarm07_6 = sensorSDSAlarm07;
82 if (sensorSDSAlarm07_6) goto <L12>; else goto <L14>;
L1 C1

```

Fig. 6. Covered CFG blocks shown in RT-Tester UI.

- [5] Software Considerations in Airbone Systems and Equipment Certification. RTCA/DO-178B, December 1992.
- [6] Stephan Eggersglüß, Görschwin Fey, and Rolf Drechsler. SAT-based ATPG for Path Delay Faults in Sequential Circuits. In *IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, pages 3671–3674, New Orleans, 2007.
- [7] Ulrich Hannemann, Tobias Hartmann, Jan Peleska, and Aliko Tsiolakis. Analysis of Interfaces and Signal Flow in the Cabin Domain. Technical report, University of Bremen, TZI, 2006. Deliverable for KATO-TP13.
- [8] International Electrotechnical Commission, Geneva. *International Standard IEC1025 - Fault Tree Analysis (FTA)*, 1990.
- [9] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis*. Springer-Verlag, London, 2001.
- [10] Alexander Miczo. *Digital Logic Testing and Simulation*. Wiley-Interscience, 2003.
- [11] Neil Storey. *Safety-Critical Computer Systems*. Addison Wesley Longman, 1996.
- [12] Verified Systems. RT-Tester 6.x – User Manual. Technical Report Verified-INT-014-2003, Verified Systems International GmbH, Bremen, 2004.