HARDWARE/SOFTWARE INTEGRATION TESTING FOR THE NEW AIRBUS AIRCRAFT FAMILIES

Jan Peleska
Center for Computing Technologies TZI
University of Bremen
jp@tzi.de

Abstract In this presentation, we describe the methods and techniques currently applied in the test of the cabin management controllers for the Airbus families A318, A340-500/600 and A380. The hardware-in-the-loop testing environment is described and we explain, how an integrated approach for software testing, hardware/software integration and system testing allows to re-use test specifications on these different levels. Moreover, the concepts for automatic test data generation and test evaluation, as well as the utilisation of generic test specifications are introduced. Our approach has been elaborated by research and development teams at Airbus Deutschland, KID-Systeme, Verified Systems International and the Author's research group at the University of Bremen. Tool support is provided by the RT-Tester real-time test tool developed by Verified Systems and the University of Bremen.

Keywords: Airbus Cabin Communication System, Software Integration Testing, Hardware/Software Integration Testing, Test Automation

Introduction

Motivation. In this article, we describe state-of-the-art concepts and novel approaches to embedded systems testing for avionics control systems. These are currently applied by Verified Systems International GmbH in collaboration with the author's research group at Bremen University for the test of the Cabin Management System CIDS, designed for the Airbus aircraft families A340-500/600 and A318. CIDS has been developed by KID-Systeme in Buxtehude, a subsidiary of Airbus Deutschland.

In the eighties and early nineties of the last century, the test of computerised avionics control systems still required a high degree of manual interaction: On system integration level, the inputs to the system under test (SUT), the checking and recording of SUT reactions, as well as the evaluation of test results were performed interactively by testing personnel, and a large portion of the test documentation consisted of paper listings filled in and maintained in a manual way. Only some external systems which interacted with the SUT in the real operational environment but were not available for SUT testing were replaced by automatic computer simulations. However, the variation possibilities for the behaviour modelled by these simulations were still limited and required a considerable amount of manual interaction. On HW/SW integration level, several manufacturers of aircraft controllers already applied *hardware-in-the-loop testing*: The stimulation of inputs to the SUT as well as the simulation of external systems were performed by computers acting as test drivers, and the SUT reactions were recorded automatically. A joint time base for stimulated inputs and recorded outputs allowed to analyse the causal relationships between the test events observed. However, the generation of inputs was mostly based on sequential test scripting languages: Each input to the SUT had to be "programmed" into the test script in a manual way, and the scripts processed these inputs on a one-at-a-time basis, which did not allow to generate complex scenarios of simultaneous inputs on several SUT interfaces. Moreover, the recorded SUT reactions had to be checked manually against the applicable requirements specifications.

Like KID-Systeme, some manufacturers changed their testing approach within the last decade towards hardware-in-the-loop testing enhanced by new techniques in order to achieve a higher degree of automation. These – often very ambitious – campaigns were motivated by the following problems:

- The growing complexity of applications increased the amount of test cases needed to achieve sufficient test coverage in a considerable way.
- The growing demand for customer-specific modifications and extensions increased the frequency of new system releases which have to be accompanied by regression testing.

An analysis of these problems performed by KID-Systeme, Verified Systems and the author's research group at the University of Bremen resulted in the integrated SW- and HW/SW-integration testing environment described in this article.

Overview. In Section 1, we give a more detailed analysis of the problems listed above and describe the testing approach and tool concept resulting from these considerations. This is the main part of this article and illustrates the main ideas which according to our understanding have been crucial for the success of our approach. The full hardware-in-the-loop test configuration is sketched in section 2. Section 3 contains the conclusion.

Background and Related Work. From the application-oriented point of view, the work presented in this article is based on the author's experience with the development of test systems for avionics controllers in the early nineties:

- Software integration test system for an earlier version of CIDS developed for the Airbus A330/340,
- Hardware-in-the-loop testing environment for a BMW-Rolls Royce aircraft engine controller.

The experiences gained from these systems resulted in the development of the RT-Tester system [18] which is used for the testing approach described in this article and for other testing projects in the fields of railway, space and automotive control systems. Reports about other industrial applications can be found in [12] (test of tramway crossing control system) and [16] (test of fault-tolerant computer for the International Space Station).

From the theoretical point of view, the problems of automatic test generation, test execution in hard real-time and test evaluation have been investigated by the author and his research team at Bremen University in collaboration with several other scientists. In [11, 13, 14] the theory for automated testing of reactive systems without timing requirements are described. The foundations of the theory are based on Hennessy's testing equivalence elaborated for process algebras with acceptance tree semantics [3]. For real-time testing, our formalisms are based on the semantics of Timed CSP (Communicating Sequential Processes) as given by Schneider [20]. Comprehensive introductions to CSP are given in [5, 17]. For the associated algorithms implemented in the RT-Tester tool we made use of a theorem about the executability of CSP specifications in hard real-time, which has been formulated by the author [15] and established in complete form by Oliver Meyer [9].

The field of test automation based on formal methods is currently investigated world-wide by several research groups. We name [1, 10, 19, 21] as a set of representative publications which also give an extensive overview of existing publications in this research area.

The examples from CIDS presented in this article have been simplified in order to focus on the underlying concepts. See [7] for a more detailed description of CIDS.

1. Testing Approach and Tool Environment

To overcome the two main problems about growing testing complexity and regression testing workload described in the introduction, five main goals were defined and implemented in the testing environment. These will be described in the subsequent paragraphs. Throughout this section, we will refer to the organisational structure of the testing environment, as represented in Figure 1.

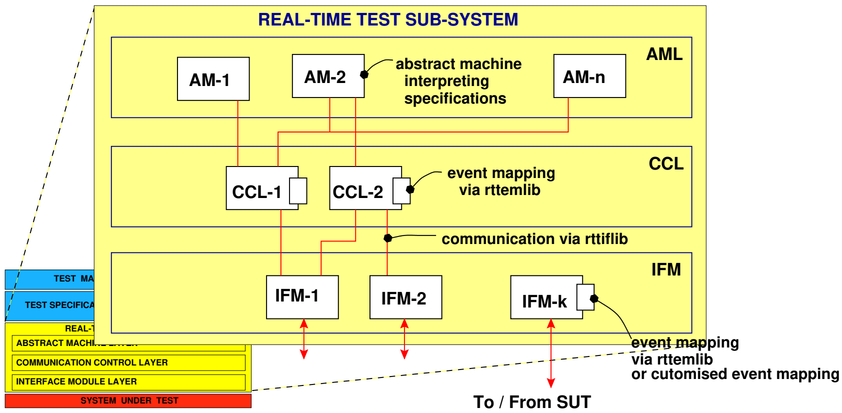

Figure 1. Organisational Structure of the Test System.

1.1 Re-use of specifications on different integration levels

Deficiencies of the conventional approach. The classical approach to embedded systems testing used different testing environments and different test specification techniques on module test, SW integration test, HW/SW integration test and system integration test level. Typically, the specifications used for module and SW integration testing referred to SW functions and data structures, while the higher integration test levels referred to events observable at various system interfaces. As a consequence, the test results achieved on these different integration levels were very difficult to compare, since the terms used in software tests had to be translated to associated terms about data passed along system interfaces. Moreover, it was impossible to re-use test specification parts on different levels without translating them into other representations.

Re-use by Interface Abstraction. The testing efficiency can be considerably increased if a single set of *interface abstractions* is used from software integration (or even module) testing level to system integration. This allows specification parts referring to system interfaces only to be re-used on all test levels. (Of course, internal software interfaces manipulated or monitored during software integration testing cannot be accessed by system integration testing.) For the RT-Tester system we choose CSP-style channel specifications for interface definition syntax on all integration levels. The abstract channel specification is mapped onto concrete interfaces that depend on the test integration level. To facilitate the setup of different test configurations, the test system is structured into three conceptual layers (see Figure 1): The *abstract machine layer AML* contains the mechanisms to create test data and check SUT responses (see the more detailed description below). This is done on the abstract level of CSP events, and each abstract machine is equipped with a CSP channel interface specification. The *interface modules (IFM)* are used to map abstract CSP events onto concrete SUT interfaces and vice versa. The *communication control layer (CCL)* relays events between abstract machines and interface modules. Whenever an interface is accessed on different test integration levels, the same CSP channel description is used: we only need to exchange the associated IFM according to each level's interface semantics.

Example: Interface from Flight Attendant Panel to CIDS Reading Lights Application. The CIDS Reading Lights application controls the switching of reading lights throughout the Airbus cabin: External systems, e. g., control panels installed for the cabin crew, will send reading lights control commands to the CIDS controller which reacts by switching the lights associated with the respective command. The interfaces involved in the operational environment have been sketched on the left-hand side of Figure 2. For the *"Reading Lights Off"* command, the CSP channel representation for these inputs to the SUT is given as

```
fap_id_t = { 1..10 }
channel FAP_key_cil_rl_off : fap_id_t
```

A test event `FAP_key_cil_rl_off.1` represents the situation where at flight attendant panel (FAP) 1, the "Reading Lights Off" key has been pressed.

For software integration testing, we observe that the Reading Lights Control software component accesses key events using a driver function call `keyEvent = getKey()`. This function is exchanged for testing purposes by a stub with identical interface, which reads the `keyEvent` data object to be returned to the caller from a shared memory buffer (see right-hand side of Figure 2). An interface module accepts abstract CSP channel events like `FAP_key_cil_rl_off.1` which have been generated by abstract machines simulating user interactions at various FAPs, transforms them into new `keyEvent` data objects which are placed into the shared memory buffer.

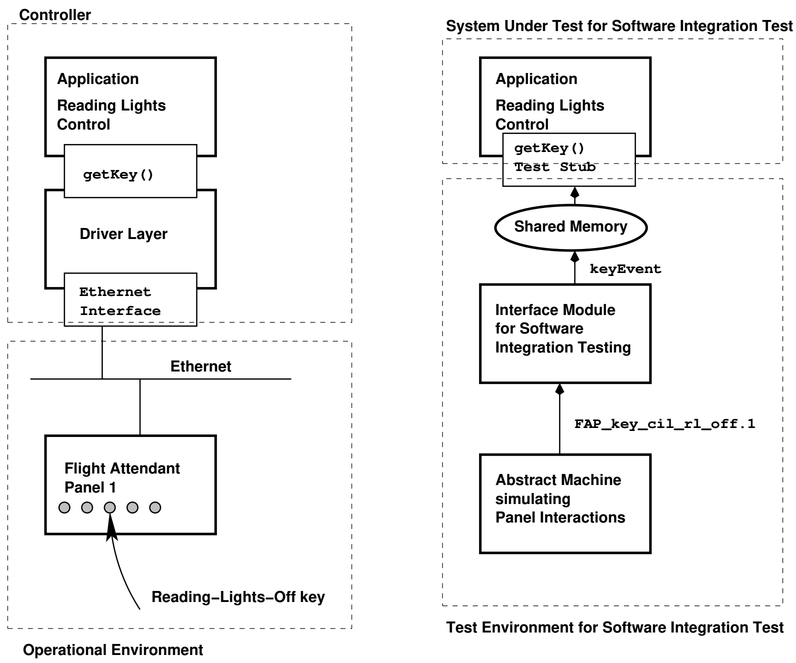

Figure 2. Interfaces in the operational system versus software integration test interfaces.

For HW/SW integration testing, we would like to utilise the same abstract machines, but have to take into account that the interfaces to be accessed on this integration level is given by an Ethernet-based protocol (see left-hand side of Figure 3). A new IFM accepts the same CSP events from abstract machines. Now the CSP events are transformed into telegrams encoding the key event, and the telegrams are transmitted using TCP/IP.

For system integration testing (see right-hand side of Figure 3), the CSP interface and the generating abstract machines still remain the same, but now another IFM transforms them into pointing device commands passed to the panel via serial link. The pointing commands have the same effect on the panel as the touch screen interactions performed by users.

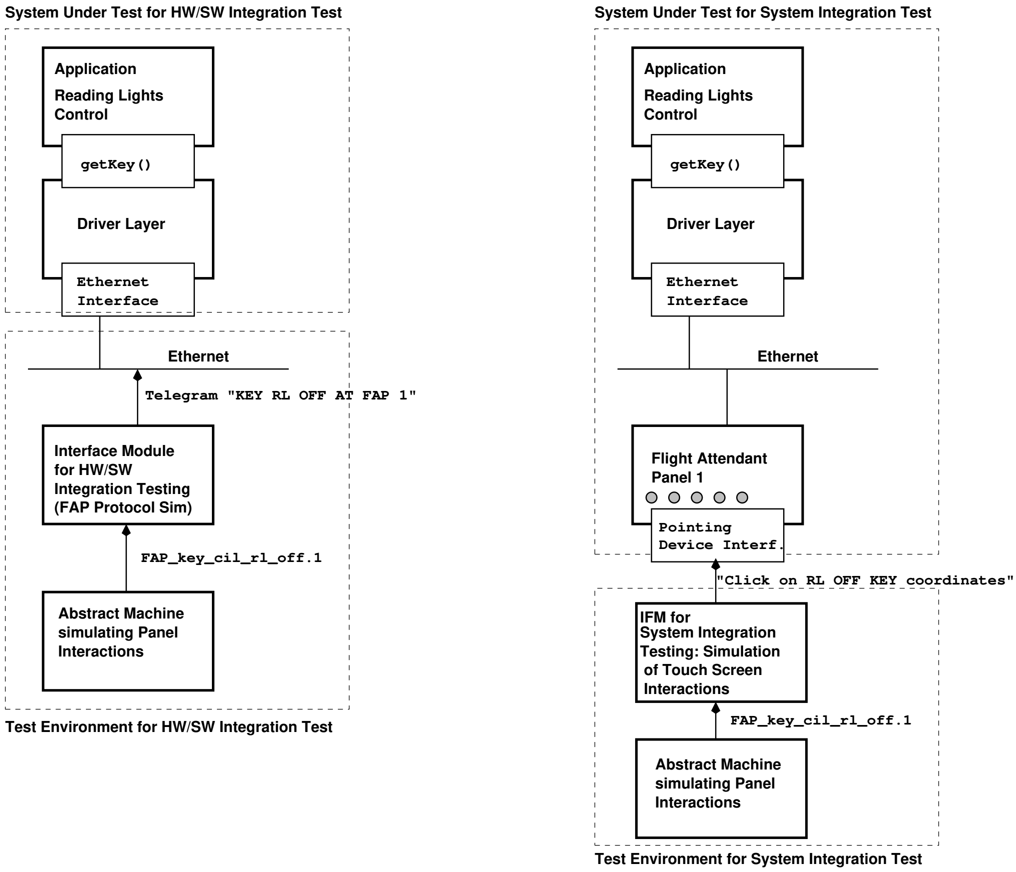

Figure 3. Interfaces for HW/SW integration and system integration test.

1.2 Automatic test data generation from specifications

Deficiencies of the conventional approach. Many severe errors in control systems are only detected when special sequences of inputs are exercised on the system. Typically, these sequences occur in real operational situations, but – due to a lack of application expertise – are overlooked or taxed as "unrealistic" by the developers and testers involved, so that these situations are neither covered properly in the design nor in the test suites. A major cause for this problem is given by the sequential programming language style of many test scripting languages, which makes the generation of combinatorial patterns for input sequences a very tedious and time-consuming task to perform.

State machine based test data generation. Instead of defining test data in an explicit way, it should be automatically created from a set of "generation rules", so that many different combinations of stimuli can be derived from a small set of rules. An appealing approach is to use state machines to specify these rules: The possible sequences of inputs and outputs which may be generated by the SUT interacting with its environment may be regarded as words of a formal language, and it is well known (see [8]) that automata can be used to check whether given words are members of a specific language. For test generation, we just turn this concept upside-down, starting with timed state machines modelling the environment behaviour to be used in a test case and interpreting a machine-readable representation of the state machine to generate legal words, that is, legal traces of inputs to the SUT which may also depend on SUT outputs and timing conditions before they are triggered. This basic approach has been investigated by several research groups (see the bibliography below) using various formalisms for timed state machines or transition systems. The results of Meyer [9] show that also in presence of real-time conditions the language generated by the timed traces (i. e. the sequence of communications tagged with time stamps) remains regular. Therefore these state machines can be interpreted without backtracking and in bounded time, so that the principle is suitable for hard real-time testing with on-the-fly test data generation.

Example: Creation of combinatorial event sequences. The following example has been developed for a situation where *"Reading Lights On"* and *"Reading Lights Off"* commands should be tested while choosing different panels for creation of the command inputs. These commands can be issued fro the flight attendant panels introduced above and additional attendant panels (AAP) residing on different flight decks at specific bus addresses and ports. The CSP process `RL_TEST` below performs the data generation in real-time, making use of two subordinate processes `SELECT_FAP_EVENT(chan)` and `SELECT_AAP_EVENT(chan)`: At first the process waits for a time interval specified by timer `rl_timer`. After this an event `FAP_key_cil_rl_on.<id>` is generated (in auxiliary process `SELECT_FAP_EVENT(chan)`), where the internal choice operator `|~|` denotes nondeterministic selection of an id parameter from the set `fap_id_t`. After another waiting interval, auxiliary process `SELECT_AAP_EVENT(chan)` selects an event `AAP_key_cil_rl_off.<d>.<b>.<a>`, where the triple (d, b, a) is chosen nondeterministically as specified by the nested `|~|` construct. After this, process `RL_TEST` starts another recursion.

In the RT-Tester system, the choice operators are evaluated to select data from the specified ranges in an automatic way. When a state is revisited, the test system selects new data tuples from the choice sets. The algorithm is designed as a combination of random choices and systematic coverage of branches which have not yet been chosen. As a consequence, `RL_TEST` will generate the more of ON/OFF events the longer it runs. Timers may be associated with random intervals, allowing to gain higher coverage in the time domain, if certain WAIT-situations are revisited.

```
channel FAP_key_cil_rl_on : fap_id_t
DECKS = { 0..1 }
ADDRESS = { 1..15 }
AAP_PORT = { 0..1 }
channel AAP_key_cil_rl_off : DECKS.ADDRESS.AAP_PORT

RL_TEST =
    WAIT(rl_timer);
    SELECT_FAP_EVENT(FAP_key_cil_rl_on);
    WAIT(rl_timer);
    SELECT_AAP_EVENT(AAP_key_cil_rl_off);
    RL_TEST

SELECT_FAP_EVENT(chan) =
    (|~| id : fap_id_t @ chan!id -> SKIP)

SELECT_AAP_EVENT(chan) =
    (|~| d:DECKS @ (|~| b:ADDRESS @ (|~| a:AAP_PORT @ chan!d.b.a -> SKIP)))
```

1.3 Automatic evaluation against specifications

Automatic generation of test data may lead to millions of communications between testing environment and SUT during a single test execution. While it is desirable to stimulate and record such large numbers of testing events, their manual evaluation will be infeasible in most cases. As a consequence, the success of the testing approach depends on possibilities for automatic evaluation. Formally speaking, correctness conditions can always be expressed by assertions about the timed traces (i. e. the sequence of communications tagged with time stamps) observed during test executions. These assertions may be expressed as logical formulae (e. g., a variant of temporal logic or trace logic) or equivalently as a collection of state machines admitting only those traces which are compliant with the assertions.

Observe that the most powerful assertion about the SUT is given by the requirements and design specifications. If they are available in a formal and machine-readable representation, the test evaluation problem is completely solved.

Observe further that for regression testing in the real-time domain, the test execution log (i. e., the recorded timed trace) of a previous test is unsuitable as a reference for regression testing: Especially in the case of complex controllers with many interfaces, the behaviour as it appears at the SUT interface will always be nondeterministic to a certain degree, for example due to timing jitters and internal scheduling decisions which cannot be observed during the test. As a consequence, a one-to-one comparison with previous execution results will nearly always fail. Instead, checking should be performed against assertions which are liberal enough to admit legal deviations in the regression run which are consistent with the applicable specifications.

1.4 Test configurations as distributed systems

A second problem with conventional test scripting languages consists in the fact that – similar to sequential programming languages – they encourage testers do elaborate test cases where one interface is stimulated or observed at a time. As a consequence, errors that only occur if certain events occur simultaneously will not be detected by the testing team. Moreover, it is often impossible to monitor large numbers of SUT output interfaces in parallel during the test execution. Under these circumstances, errors which are caused by instabilities at interfaces which should *not* be affected by certain operations are often overlooked. These observations motivate a testing approach where the testing environment is designed as a distributed system and where all forms of parallelism that are possible in the real operational environment can occur in the testing environment, too. This approach is modelled by a network of timed state machines operating in parallel. Each machine may act on certain SUT input interfaces, monitor and check SUT outputs and communicate with other state machines in order to exchange information about the current status of the test execution.

1.5 Generic test specifications

Today, many control systems can be configured by tables defining "software switches" which influence the behaviour of the system. For product certification, it is mandatory to demonstrate that the system handles a large variety of configurations correctly. From an object-oriented point of view, the control software may be regarded as a generic class. A concrete object of this controller class is generated by instantiating the software for a concrete configuration table. This observation leads to the requirements that test specifications should be generic as well: The configuration dependent rules for test data generation and evaluation are specified using generic parameters equivalent to the ones evaluated by the control software and are instantiated by the same tables. This allows to use a single set of test specifications for all test cases checking one generic aspect of SUT behaviour in presence of configuration data variations.

Example: Use of generic parameters in test specifications. The CIDS configuration table allows to configure the aircraft cabin zones which are affected by a *"Reading Lights Off"* key dependent on the specific panel where the key was pressed. For a specific test case with fixed configuration table T, the test system compiles T into parameter definitions using mathematical data types (constants, sets, sequences and combinations thereof). For *"Reading Lights Off"* keys, we use sets for each cabin zone z, containing the AAP identifications (deck, address, port) of those panels where the reading lights off key has an effect on z: Generic parameter

```
GENERIC_RL_RESET_AAP_ZONE_1 = { 0.1.1, 0.2.1 }
```

states that reading lights in zone 1 are switched off by *"Reading Lights Off"* keys pressed on AAPs with $(deck, address, port) = (0,1,1)$ and $(deck, address, port) = (0,2,1)$. The checker specifications are modularised by the cabin zone structure: `RLCHECKER_ZONE_1` checks timeliness and correctness of reading light state changes in zone 1:

```
RLCHECKER_ZONE_1 =
  . . .
  AAP_key_cil_rl_off?d.b.a
       -> ( if ( member(d.b.a,GENERIC_RL_RESET_AAP_ZONE_1) )
            then ( setTimer.1 ->
                     ...  wait for all reading lights in zone
                          1 to be switched off before timer 1
                          has elapsed ...)
            else ... disregard this AAP key ... )
  . . .
```

Whenever an AAP key event is simulated by the test data generator (such as process `RL_TEST` introduced above), the event is registered by all zone checkers as an input statement `AAP_key_cil_rl_off?d.b.a`. Using the `member()` condition in the `if`-statement above, the checker determines whether the panel affects the local zone. If this is the case, the checker sets a timer and waits for all reading lights to be switched off within the admissible time interval. If the panel does not affect the local zone, it is just ignored by the checker.

For a new configuration table, the checker specification is recompiled with the new definitions of `GENERIC_RL_RESET_AAP_ZONE_1` and other generic parameters. The specification source code remains unchanged. Note that the real checkers are also generic in the cabin zone number, so that a single source may be used for all zone-specific checkers and all configuration table settings.

2. Hardware-in-the-Loop Test Configuration

After having explained our testing approach on a conceptual level, we will now describe the test configuration for HW/SW integration testing in more detail (see Figure 4).

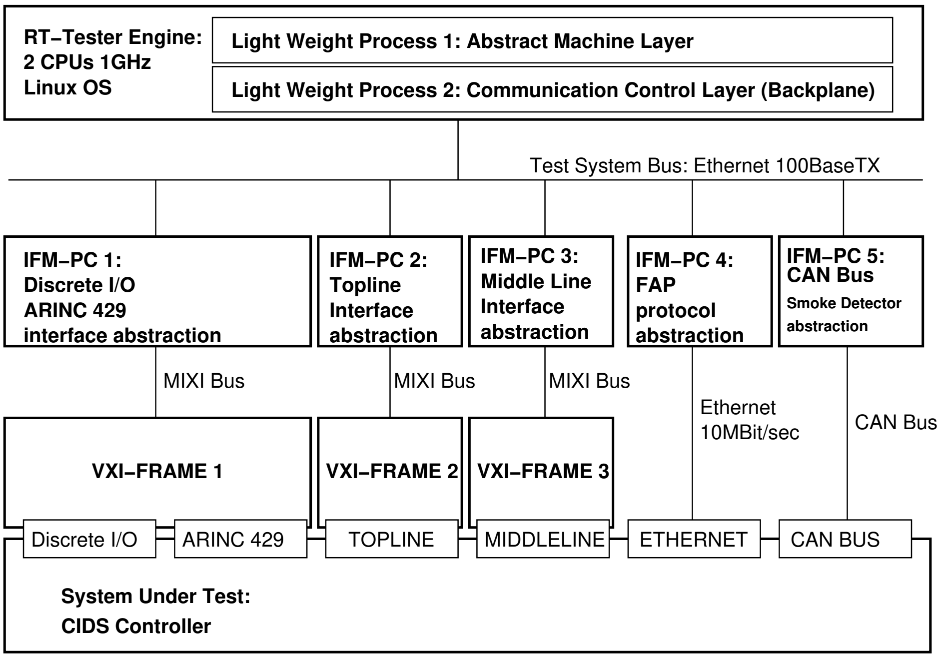

Figure 4. HW/SW integration test configuration.

The System Under Test is now given by the CIDS controller alone; panels and other CIDS devices outside the controller are not part of this configuration. The CIDS controller commands a variety of interfaces: Discrete I/O is used to switch external devices on/off (e. g., the flight warning buzzer in the cockpit) and to receive discrete state information (e. g., state of landing gears). The ARINC 429 bus is used to communicate with other controllers (e. g. , the environmental control system responsible for air conditioning). Topline and Middleline are proprietary busses developed by KID-Systeme which are used to control external devices (such as reading lights or AAPs) via intelligent bus interfaces. The Flight Attendant Panels (FAP) described in the examples above are connected to the CIDS controller via Ethernet. The CAN bus interface (A318 only) is used for communication with smoke detectors.

In the HW/SW integration test environment, VXI systems are used to read/write date on the first 4 interfaces described above. The interface module software is integrated in PCs acting as front ends to the VXI systems, so that only interface boards and low-level drivers are allocated inside the VXI frames. For Ethernet and CAN bus, standard PCI cards are used.

Test data generation and on-the-fly evaluation is performed in the test engine. To exploit the CPU power available in an efficient way, the RT-Tester system allows to allocate abstract machines and communication control layer on a configurable number of light weight processes (LWPs). Typically, each LWP will be executed on a separate CPU. Within each LWP, scheduling is performed by a specific cooperative multi-tasking technique developed for RT-Tester in order to optimise real-time performance: Context switches between abstract machines or CCL processes are implemented on user thread level and do not require the participation of the underlying operating system. In particular, an increasing number of AMs participating in a test execution will not compromise the performance since the effort needed for context switching is very close to that of a C function call.

Simple tests only need a small number of AMs, but more complex tests involving several applications (e. g. emergency evacuation system tested in cooperation with the telephone and passenger address systems) may utilise several hundred AMs, each stimulating or checking a specific logical aspect of the whole test execution.

For the whole set of software and HW/SW integration tests, about 3000 different generic CSP specifications are used. They access over 1500 different logical interfaces, which may refer to global variables or function call interfaces in software tests or to specific logical data items passed along one of the HW interfaces shown in Figure 4, when HW/SW or system integration testing is performed. The number of different discrete events which are passed along these logical interfaces and either generated or checked in at least one test case is about 1.1 million.

3. Summary

3.1 Conclusion

In this article, we have presented the underlying concepts and architecture of a testing environment for the Cabin Management controllers in the new Airbus A318, A340-500/600 and A380 aircrafts. The distinguishing features in contrast to conventional testing approaches were (1) re-usability of tests on software integration test, HW/SW integration and system test level, (2) automatic test data generation and test evaluation based on networks of timed state machines and (4) utilisation of generic test specifications.

The main advantages of our approach are (a) Automatic test data generation often generates situations where errors are uncovered but which typically would not have been investigated in manual tests. (b) Identification of error causes is facilitated because tests can be replayed with small modifications on lower integration levels: If a system test uncovers an erroneous situation, it can be replayed on software integration level using auxiliary debugging tools for pinpointing software bugs. (c) The effort needed for regression testing is considerably reduced because tests can be automatically replayed and correctness checks are based on specification that do not reject legal deviations of the regression test execution from the reference test results. (d) The overall testing effort is considerably reduced because of the automatic generation/evaluation and documentation facilities and the re-use of concept, interfaces and specification parts on different integration levels.

3.2 Related Activities and Future Work

Due to the usual space limitations, several aspects of ongoing and future work which are related to the material presented above can only be mentioned briefly:

To improve the hard real-time testing capabilities, a new test engine concept based on PC clusters with a customised hard real-time variant of the Linux operating system is currently in its trial phase. All relevant hardware interfaces needed for testing avionics controllers are directly integrated as PCI boards in the test engine cluster. The cluster communication is performed via FireWire (IEEE 1394) or Myrinet. This activity is performed within the VICTORIA project, which is part of the current European R&D Framework Programmes and focuses on the development of novel technologies for avionics control and its validation, verification and test.

For software integration testing, a large portion of the overall testing effort has to be invested into interface module development, because often a large number of software interfaces has to be accessed during different software integration stages. It is currently investigated how the IFM generation can be (at least partially) automated by generating the abstraction mapping between software interfaces and CSP channels in an automatic way.

For practical testing in an industrial setting it is important to know how the tool-based testing approach fits into the project development approach defined by the underlying V-Model and other applicable standards. A publication elaborating these questions for the Avionics Software V-Model RTCA/D 178B [2] and the IEEE standard [6] on test documentation is currently under preparation.

The reader may have noticed that in the test setting described in this article, interfaces were considered to be discrete. This is true for the CIDS controller. However, in many embedded systems test applications, time-continuous data has to be generated and checked as well. To this end, we enhance the abstract machines introduced above by global real-valued state variables which may change "continuously" over time according to differential equations or similar specifications of time-continuous evolutions. The underlying semantic concept is given by Hennessy's *Hybrid Automata* [4].

Acknowledgments

The project described in this article would be unmanageable without the help and initiative of many people. The author would like to thank Manfred Endreß, Thomas Fritz, Manfred Kühnel, Sascha Marckwardt and Ute Meyer (KID-Systeme), Cornelia Zahlten, Christof Effkemann, Oliver Meyer, Raymond Scholz, Hauke Steenbock, Kai Thomsen (Verified Systems International), Markus Dahlweid, Dirk Meyer and Aliki Tsiolakis (Bremen University) for reliable and stimulating cooperation in this – sometimes rather challenging – project.

Last, but not least, the author would like to express his gratitude to the organisers of the TestCom2002 for granting him the possibility to deliver this presentation.

References

[1] Cardell-Oliver, Rachel. (2000). Conformance Testing of Real-Time Systems with Timed Automata. In *Formal Aspects of Computing.* (12):350-371

[2] RTCA DO178B: *Software Considerations in Airborne Systems and Equipment Certification.* (1993).

[3] Hennessy, M. C. (1988). *Algebraic Theory of Processes.* MIT Press.

[4] Henzinger, Th. A. (1996). in *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pp. 278-292.

[5] Hoare, C.A.R. (1985). *Communicating sequential processes.* Prentice-Hall International, Englewood Cliffs NJ.

[6] IEEE Std 829-1998. *IEEE Standard for Software Test Documentation.* The Institute of Electrical and Electronics Engineers, New York.

[7] KID-Systeme. *CIDS Digital Cabin Management System.* Information available under `http://www.kid-systeme.de`.

[8] Lewis, Harry R. and Papadimitriou, Christos H. (1981). *Elements of the Theory of Computation.* Prentice-Hall International, Englewood Cliffs, New Jersey.

[9] Meyer, Oliver. (2001). *Structural Decomposition of Timed CSP and its Application to Real-Time Testing.* Dissertation, Universität Bremen, FB 3.

[10] Nielsen, Brian. (2000). *Specification and Test of Real-Time Systems.* Ph. D. Thesis, Department of Computer Science, The Faculty of Engineering and Science, Aalborg University, Publication No. 12.

[11] Peleska, Jan and Siegel, Michael. (1996). From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock (Eds.): FME '96: Industrial Benefit and Advances in Formal Methods. LNCS 1051, Springer-Verlag, Berlin Heidelberg New York 538-556.

[12] Peleska, Jan. (1996). Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. (invited keynote), In M.-C. Gaudel and J. Woodcock (Eds.): FME '96: Industrial Benefit and Advances in Formal Methods. LNCS 1051, Springer-Verlag, Berlin Heidelberg New York pp. 39-59.

[13] Peleska, Jan. (1997). *Formal Methods and the Development of Dependable Systems.* Habilitationsschrift, Bericht Nr. 9612, Dezember 1996, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel.

[14] Peleska, Jan and Siegel, Michael. (1997). Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal* 19:53-77.

[15] Peleska, Jan. (1998). Testing Reactive Real-Time Systems. Tutorial, held at the FTRTFT '98. Danmark Technical University, Lyngby.

[16] Peleska, Jan and Buth, Bettina. (1999). Formal Methods for the International Space Station ISS. In E.-R. Olderog, B. Steffen (Eds.): Correct System Design, Springer LNCS 1710, pp. 363-389.

[17] Roscoe, A. W. (1998). *The Theory and Practice of Concurrency.* Prentice-Hall International, Englewood Cliffs NJ.

[18] Verified Systems International GmbH. (1998). *RT-Tester Test Automation System.* Information available under `http://www.verified.biz`.

[19] Sadeghipour, Sadegh. (1998). *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications.* Schriftenreihe Forschungsergebnisse zur Informatik, Bd. 40. Verlag Dr. Kovaĉ, Hamburg.

[20] Schneider, Steve. (1995). An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213.

[21] Tretmanns, Jan. (1992). *A Formal Approach to Conformace Testing.* PhD Thesis, University of Twente, Haag.