Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002
©2002 Society for Desing and Process Science

Formal Methods for Test Automation - Hard Real-Time Testing of Controllers for the Airbus Aircraft Family

Jan Peleska
TZI, Universität Bremen, P.O. Box 330440, D-28334 Bremen, and
Verified Systems International GmbH, Bremen
jp@tzi.de

ABSTRACT: In this article, a collection of major problems to be solved for automated testing of embedded hard real-time systems is discussed. It is indicated which solutions are available. Architectural aspects of test automation systems and approaches for automated on-the-fly test evaluation are elaborated in detail. Practical examples refer to avionics control systems for the Airbus aircraft families which have been tested by the author's research team at Bremen University in collaboration with Verified Systems International.

I. Introduction

A. Motivation: Formal Methods and Testing

This article discusses issues about automated testing of reactive hard real-time systems. Systems are called *reactive* if they are prepared to interact continuously with their operational environment. Typically, control computers are designed according to the reactive system paradigm, since they should always be prepared to adjust ongoing control activities to user interactions or feed backs from the environment they control. The term *hard real-time* is used in the sense that the behavioral correctness of a system under test (SUT) also depends on the time intervals (so-called *deadlines*) when expected SUT outputs occur (a more detailed discussion of correctness conditions is given in Section II-E). We consider formal verification and testing as complementary activities that are both part of the quality assurance process. Ideally, the product-related quality assurance tasks would be split between formal verification and testing as follows:

- Logical correctness properties of requirements specifications, design specifications and code should be formally verified.
- The proper integration of software, firmware and hardware should be tested.
- The reliable operation of controllers should be tested by *built-in test equipment* which monitors operations and performs on-the-fly checks of compliance with the specified behavior.
- Completeness properties of requirements specifications which cannot be deduced from other reference specifications should be validated by a combination of formal verification and simulation, that is, testing on symbolic specification level.

For today's reactive real-time systems – at least when they perform safety-critical or mission-critical control tasks – a high degree of automation is required. Otherwise it would be infeasible to achieve the necessary degree of test coverage and to perform regression testing on new product revisions within acceptable time/cost margins. Moreover, timing conditions often require observational resolutions from a few milliseconds down to micro seconds, so that neither the generation of inputs to the system under test nor the measurement of SUT reactions could be performed manually with sufficient precision. As a consequence, testing has to be based on formal specifications which can be interpreted by computers in an automatic way.

B. Background: Testing Avionics Controllers for the Airbus Aircraft Family

The theoretical concepts described in this article have been implemented in the test automation tool RT-Tester [26] developed in cooperation of Verified Systems International GmbH and the author's research team at the University of Bremen, TZI. The tool has been applied in practice since the early nineties starting with tests for tramway control systems [18]. Currently, the tool is mainly used in the fields of railway control systems, space systems, avionics systems and telecommunications, see [3], [4], [22], [23], [24], [29].

The examples presented in this article are derived from experiences with avionics control systems tested by Verified Systems for Airbus Deutschland. The systems are integrated in the Airbus A318 and A340-500/600. At present, test configurations for A380 control systems are designed and implemented.

C. Related Work

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 [17], [19], [20] the theory for automated testing of reactive systems without timing requirements are described. The foundations of the theory are based on deNicola's and Hennessy's testing equivalence elaborated for process algebras with acceptance tree semantics [9]. For real-time testing, our formalisms are based on the semantics of Timed CSP (Communicating Sequential Processes with Time, TCSP) as given by Schneider [28]. Comprehensive introductions to CSP are given in [11], [25]. 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 [21] and established in complete form by Oliver Meyer [15].

The field of test automation based on formal methods is currently investigated world-wide by several research groups. We name [6], [16], [27], [30], [5] as a set of representative publications which also give an extensive overview of existing publications in this research area.

D. Overview

In this contribution, we describe problems which "naturally" arise in the context of testing for embedded, possibly safety-critical, controllers and show how these problems can be solved and implemented by means of approaches based on Formal Methods. A survey of these problems is given in Section II. Here we state six tasks to be solved for efficient hard real-time testing, which are important according to our understanding of theoretical foundations and testing in practice. In the same section, references to research activities and known solutions elaborated by various research groups for the problems described are listed. Analysis of the problem survey of Section II leads to the proposal of a *generic design* for test automation systems applicable in the field of embedded control systems. This design is described in Section III, and we indicate which design components should be responsible for implementing solutions to the problems stated before. In Section IV, solutions for automated test evaluation are discussed in more detail. Section V contains the conclusion.

II. A Problem Survey: Six Requirements for Hard Real-Time Testing

In this section, a number of major problems will be described which we consider as crucial for the development of trustworthy and efficient test automation systems.

A. Re-use of Test Specifications on Different Testing Levels

In this article, the term *test specification* is used to denote the collection of (formal or informal) descriptions that are necessary to perform test executions with well-defined objectives. Typically, a test specification consists of (see [12] for a more detailed introduction of testing terminology)

- Test procedure: specification how to perform a test execution in a step-by-step manner,
- Test data: specification of input data to be passed to the SUT and of the conditions when each input should be made,
- Expected results: specification of the correct SUT behavior during a test execution with given test procedure and test data.

A systematic approach to reactive systems testing requires to perform tests on different *levels* allowing to focus on complementary aspects of SUT behavior. The usual levels are

- Module tests: test of isolated functions or methods in a software test harness,
- Software integration tests: test of cooperating software components (class instances, threads, processes, software layers,. . .) on the target or in a host environment,
- Hardware/software integration tests: hardware-in-the-loop test of the complete software integrated on the target hardware,
- System integration tests: test of cooperating sub-systems, possibly consisting of networks of communicating controllers and original components as peripherals.

In most conventional testing approaches each test level uses its own data structures – from programming variables on module level to data bus telegrams on system test level. As a consequence, specifications cannot be re-used on different levels and test execution data obtained on different levels are extremely hard to compare.

A solution to this problem consists in using an abstract formal language for test specification and for the representation of test executions: Interfaces to the SUT are associated with abstract names and data is represented in a syntax which is easy to understand and independent on specific interface format requirements. During test executions, test system components perform the transformation between abstract representations and concrete interface data (*refinement*) and vice versa (*abstraction*). This concept of *interface abstraction* is elaborated in more detail in [23], [24].

B. Test configurations as distributed systems

Traditionally, test specifications were written as sequential scripts describing the inputs to be written to SUT interfaces at certain points in time and the expected SUT outputs. These sequential scripting techniques have two major disadvantages:

- The linear script does not reflect the architecture of the SUT and its surrounding environment. As a consequence, the relationship between script and system architecture is difficult to explore.
- Combinatorial patterns are tedious to generate in a linear script.

To avoid these problems, we consider test configurations as distributed systems where test data generators are structured according to the architectural design of the SUT and its environment. Test evaluation is also performed by different test system components cooperating in parallel and structured according to the distribution of capabilities as described by the SUT design.

These considerations have led to a generic design for test automation systems consisting of a network of components which are described in more detail in Section III.

C. Time Measurement in a Distributed Testing Environment

Checking the correctness of reactive systems timing behavior requires precise measurement of communication events taking place between SUT and test environment. In particular, time measurements should be made as close to the interfaces as possible, since otherwise small measurement errors are propagated through each test execution, accumulate and may lead to a rejection of SUT executions even though they are compliant with their timing requirements. As a consequence, computers in the testing environment need a common time basis which can be synchronized with time stamps generated on interface boards for high-precision hardware-in-the-loop testing. Furthermore, feed back from interface boards may be required to indicate when an input to the SUT really occurred on the hardware interface.

D. Automated test data generation

A major criticism about conventional testing tools is that they require to develop test scripts where every input to the SUT is written down explicitly. This approach is unsuitable for long-duration testing where millions of inputs may be involved. The solution to this problem is to use formal specifications describing the generation rules and executable interpreters performing the actual data generation according to these rules. Especially in the case of nondeterministic SUTs these generators should be able to produce appropriate test data on-the-fly: The next input to the SUT often depends on the the SUT outputs produced so far. An "unexpected" SUT reaction might therefore invalidate the test data produced before the test execution started. Instead, on-the-fly generators may adjust to such a reaction and produce the following inputs accordingly. Generators fulfilling these objectives can be developed using suitable encodings (e. g. as transition graphs) that can be easily evaluated.

As a specification basis for test data generation, two options are available:

- If the SUT should be capable to run properly in arbitrary environments, test data has to be derived from the SUT specification itself, since no specific environment description exists. See [16] for suitable test generation solutions applicable in this case.
- If the operational environment is well-defined and controls most of the SUT activities, it may be more appropriate to use specifications of possible environment behaviors as a basis for test generation. Test generation is now very close to simulation of environment behavior. This approach has for example been investigated in [20], [19].

E. Automated test evaluation

E.1 Specification of Correctness Criteria

While correctness criteria for sequential software modules can be specified as relations between pre- and post-states of programming variables, more complex criteria have to be elaborated for (generally non-terminating) reactive hard real-time systems. Initially, the related specification formalisms have been developed in the software engineering and formal methods communities to support rigorous development and verification. In the context of testing, formalisms to specify correctness criteria are needed for the definition of expected results.

Correctness criteria for reactive hard real-time systems refer to

1. Discrete data transformations,
2. Causality properties: sequencing and synchronization of discrete inputs and outputs,
3. Refusal properties: For systems following the synchronous communication paradigm (e. g., OCCAM software or Ada software communicating via rendezvous mechanism), correctness properties also refer to the system's capability to accept or refuse input channels in specific states,
4. Timing, specifically deadlines required for discrete SUT outputs,
5. Time-continuous properties: piecewise continuous (often differentiable) changes of analog data controlled by the SUT, for example by using analog actuators,
6. Liveness properties: aspects related to discrete input/output sequences of infinite length (e. g. fairness, see [2]),
7. Asymptotic time-continuous properties: properties of real-valued observables related to boundary values, integrals etc.,
8. Non-functional properties: Reliability, availability, maintainability, security, performance, usability etc.

Synchronous communication played a major role in initial research activities which related testing to the semantic characterization of processes [9]. It has been shown by several authors (see the literature collected in [5]) for various synchronous formalisms that implementable algorithms for this so-called refusal testing exist and can be applied for practical testing purposes. However, refusal testing is of minor importance for HW/SW integration and system integration testing, because most hardware communication interfaces used in practice operate in a non-blocking mode. Even on the level of software testing refusal testing is seldom required, since programmers still prefer to use buffered non-blocking communication to synchronous mechanisms.

Liveness properties are usually investigated with formal verification techniques instead of testing, since the justification that a test execution of finite duration implies properties about infinite I/O sequences is usually as complex as a full proof of the property. The same holds for asymptotic time-continuous properties.

Non-functional requirements require different (e. g., statistical) specification and evaluation methods which are outside the scope of this paper. They often require to analyze complete test executions, so that only *a posteriori* evaluation is possible.

As a consequence, we consider the correctness properties 1, 2, 4, 5 to be the most important ones in the context of reactive hard real-time systems testing.

E.2 Automated Evaluation of Test Execution Against Expected Results

When comparing the different techniques how the above correctness properties are represented in expected results specifications, four main approaches can be distinguished

- The sequential test script approach defines correctness properties using programming variables to store and evaluate the values of recent input and output data as well as timing information.
- The record-and-replay approach requires to analyze one test execution manually. If this turns out to be correct, it is stored as the "golden reference" and any regression test execution is compared against the one originally recorded.
- The SUT specification approach aims at using the same formal specification of the SUT behavior which has been used for the SUT development. The sequence of discrete I/O events, their timing and the time-continues changes of data observed during a test execution are checked against SUT specification.
- The assertion approach defines a catalog of correctness properties which should be fulfilled by the SUT. Not unlike Use Cases in UML, this catalog is consistent with the full SUT specification but usually less complete.

The sequential test script approach operates on programming language level: Correctness properties are just specified as boolean expressions on programming variables. A definite advantage of this approach lies in the fact that the testing environment is quite similar to the programming environment, which is very attractive when only software tests are performed. Moreover, there is no need to learn a new test specification language. However, this technique has several major draw backs: First, properties referring to the history and the timing of I/Os observed require to introduce complex data structures to store the history information. It is necessary to program algorithms evaluating these data structures in order to check whether the given correctness property holds. If data structures and algorithms are programmed in a naive way using loops over all elements of the I/O sequence, the algorithm will need more execution time, the longer the test runs. As a consequence it will be unsuitable for on-the-fly evaluation even in soft real-time. Second, since properties are defines as expressions over data structures of the programming language, it may become very difficult to relate the programmed evaluation functions to the correctness conditions listed in the SUT specification using more abstract description formalisms.

The record-and-replay technique has the advantage that – apart from defining acceptable tolerances for timing deviations and time-continuous data – it is not necessary to specify any correctness conditions at all, since one evaluation was performed manually and others just say "it's still the same as before". Here, the main problems consist in two facts: First, regression tests with completely unchanged requirements are far less frequent than project managers would like them to be. As a consequence, the "golden reference" becomes soon worthless and a new manual evaluation has to be done. Second, the simple comparison technique is insufficient if the SUT may legally show nondeterministic behavior at its interface: It is often the case that – due to internal scheduling conditions or slight timing deviations in the test replay – unrelated outputs are produced by the SUT in different order for each replay. The test execution logs will therefore differ in the sequence of I/Os, and the comparison to the original reference fails. Improving the comparison technique by defining partial orders specifying related inputs and outputs turns out to be just as time consuming as the elaboration of proper formal specifications.

The SUT specification approach is very attractive, because no additional effort is required to develop expected results specifications. If the law "specification change precedes implementation change" is strictly enforced by project managers and quality assurance, an updated version of the system will be immediately testable. The major problems with this approach are the following: (1) To use the SUT specification for automatic test evaluation it has to be interpreted by the test computer, and therefore it must be completely formal. To the author's knowledge, no embedded real-time system of significant size has ever been completely specified in a formal way *and* really been used in practice. (2) Checking all logical conditions contained in the SUT specification can be very time consuming, so test designers usually focus on the correctness criteria which are relevant for the objective to be demonstrated by a specific test case. (3) Experience shows that many SUT errors are uncovered because test designers develop expected results specifications of their own. Theoretically, the expected results should be consistent with the SUT specification and would not contain any additional information. In practice, it turns out that this redundancy frequently helps to uncover implementation errors which are due to misinterpretations of the SUT specification.

The assertion approach encourages test designers to derive their own set of correctness properties which are relevant for a certain test objective. The approach can be applied both to formal and informal SUT specifications, leading to the redundancy of SUT specification and expected results specification activities performed by development and test teams, respectively. Care has to be taken that the set of assertions applicable to demonstrate a test objective is really complete and consistent with the original SUT specification. This requires an additional verification activity, which is part of the *review and analysis of the test cases, procedures and results* process required according to the development standard RTCA DO178B [8, 6.3.6] which is applicable for software development in airborne systems.

Example. A test objective of the initial HW/SW integration tests for a safety-critical Airbus controller was to check whether the discrete output driver mapped discrete data onto the correct output lines. The first assertion checked was *"A state change of discrete variable $v(i)$ from 0 to 1 leads to a change from LOW to HIGH at discrete output line i and vice versa."* This assertion could be checked without any errors. Fortunately, also a second assertion was checked during the test: *"A state change of discrete variable $v(i)$ does not affect any other output line than i."* It turned out that for a small number of discrete variables $v(i)$, the initial driver version illegally changed the state of other output lines $j \neq i$ in parallel to i.

For automatic evaluation of expected results they have to be specified in a formal way which can be (compiled and) interpreted by a test computer. It is desirable that this interpretation can be performed at least in soft real-time, so that deviations of the observed SUT behavior from the expected results will be uncovered *on the fly*. This avoids waiting for the completion of test executions possibly running over many ours, while errors already occurred during the first minutes of the execution.

Apart from using standard programming languages, the following techniques may be distinguished:

- Model based expected results specifications use formal executable models which input the timed trace of discrete I/Os and the values of time-continuous observables recorded during a test execution and signal errors as soon as the expected result is violated by the execution. Typical models are networks of parallel communicating state machines or processes from a process algebra. Henzinger's Hybrid Automata [10] allow to specify correctness properties of both discrete I/Os (using networks of automata) and time-continuous observables (using differential equations/inequalities, which may change with the state transitions performed by the automata). In the RT-Tester system, the TCSP [28] process algebra is currently used to specify hard real-time properties about discrete input/output interfaces. Hybrid Automata specifications – the evaluation of time-continuous aspects has to be supported by numerical libraries supplied by the user or by standard tools – are currently implemented.
- Implicit expected results specifications use predicates to be fulfilled by the timed traces and time-continuous observables. For time-continuous observables, predicates are the same as for Hybrid Automata. For the specification of assertions about timed traces, either trace logic (see, e.g. [11]) or a variant of temporal logic may be used.

In Section IV, a model-based test evaluation algorithm developed by Oliver Meyer [15] and the author is presented. This algorithm is suitable for on-the-fly evaluation in soft real-time. It supports both the SUT specification approach and the assertion approach and can be implemented on a network of checkers according to the distributed architecture for test automation systems described in Section III.

F. Generic test specifications

Today, many embedded control systems can be configured by tables defining "software switches" that influence the behavior of the system. This leads to an additional test objective which is to show that the system under test operates correctly for all – or, more realistic, for a large variety of – configuration table settings.

To develop explicit specifications of test data and expected results for each of these configurations would increase the number of specifications to be written and maintained during the testing process in a considerable way. An alternative approach is motivated by concepts from the field of object orientation: The controller may be regarded as a collaboration of generic classes which are *instantiated* using specific configuration data, resulting in an executable system. This suggests to elaborate *generic test specifications* which can be instantiated with the same data, resulting in executable tests "tailored" for the specific SUT behavior to be expected for this instantiation. Further details about generic test specifications are given in [23].

Example. For the test of the Airbus A340-500/600 Digital Cabin Management System CIDS [13], about 3000 generic specifications – each describing a test data generation procedure or expected results referring to specific functional requirements – were used in about 300 HW/SW integration test cases. On average, each specification was instantiated with 5 different configurations of the CIDS controller. Therefore, the availability of generic specification techniques reduced the overall amount of test specifications to be written and maintained from about 15000 to 3000 specification files.

III. A Generic Design for Test Automation Systems

As a first result of the problem analysis above, we propose a generic design for test automation systems in the field of reactive systems. Figure 1 shows a layered architecture which will be discussed in the subsequent paragraphs.

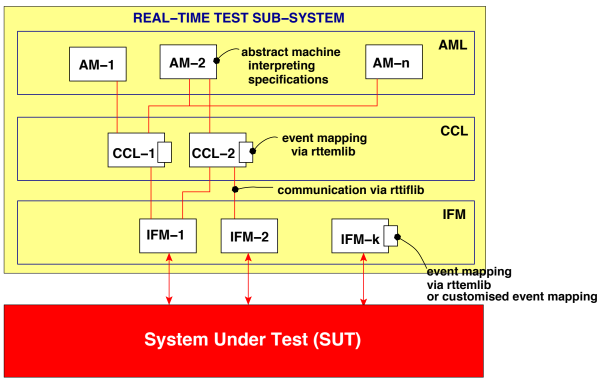
REAL–TIME TEST SUB–SYSTEM
AML
AM–1
AM–2
abstract machine interpreting specifications
AM–n
CCL
CCL–1
CCL–2
event mapping via rttemlib
communication via rttiflib
IFM
IFM–1
IFM–2
IFM–k
event mapping via rttemlib or customised event mapping
System Under Test (SUT)

Fig. 1. Generic design for test automation systems.

Abstract Machine Layer (AML). The upper test system layer contains a network of parallel *abstract machines* which are responsible for test data generation and on-the-fly test evaluation on an abstract level: Instead of directly operating on concrete interface data, abstract machines use the interface abstractions discussed in Section II-A. As a consequence, the algorithms used for test data generation and checking do not depend on the concrete SUT interface format, but are re-usable for all types of interfaces.

Test specifications are collections of formal descriptions how to generate and evaluate test data and how to control the test execution in real-time. They refer to interface abstractions and – after compilation into suitable internal representation – can be interpreted by the abstract machines. As a consequence, all test specifications are re-usable, as long only the interface format changes, but the SUT semantics – i. e. its expected behavior as seen on the abstract level – stays the same. Typically, this property is applied for re-use of test specification parts on SW integration, HW/SW integration and system integration test level.

We further suggest to structure the AML into abstract machines acting as

- *Test generators*,
- *Checkers* (or test evaluation components, also called "test oracles"),
- *Test coordinators*.

Test generators can often be designed as simulators of the real components interacting with the SUT in the operational environment. In contrast to this, checkers often need global information about several environment components and the SUT, in order to decide whether the observed SUT behavior is correct. Moreover, generators and checkers can be re-used in different situations. In addition, the integrated specification of test data generation interleaved with expected results descriptions often becomes highly complex due to combinatorial effects which have to be taken into account, so that a separation of concerns into generators and checkers operating in parallel often facilitates the test specification process in a considerable way.

Test coordinators have the task of synchronizing other abstract machines – mostly test generators – for the purpose of jointly simulating a specific type of behavior, such as normal or exceptional behavior.

Example. The Airbus A318 *smoke detection facility (SDF)* collects and evaluates information from smoke detector sensors and relays associated alarm messages to other controllers in the aircraft. When polled by the SDF controller, each detector will answer with state information like *"ok"* / *"smoke detected"* / *"sensor failure"*. For HW/SW integration testing of the SDF, each smoke detector is simulated by an abstract machine acting as test generator. Each abstract machine can generate all possible behaviors of a smoke detector. Running in parallel during a test execution, different behaviors can be simultaneously generated by these abstract machines. This helps to achieve high test coverage without specifying the combinatorial possibilities in an explicit way. However, in order to test a *normal behavior situation* where all smoke detectors respond within the correct time interval and with an *"Ok"*-value, an abstract machine acting as test coordinator signals *"show normal behavior"* to all test generators. For these coordination purposes, auxiliary abstract events are used which do not map onto SUT interfaces but are only exchanged between abstract machines.

Interface Module Layer (IFML). The *Interface Modules (IFMs)* assigned to this layer perform the tasks of refinement and abstraction suggested by the re-use problem described in Section II-A. IFMs are used to map abstract CSP events and time continuous variables given in standardized format onto concrete SUT interfaces with specific data formats and vice versa. For checking hard real-time properties of the SUT, IFMs associate time stamps u to each I/O passed along the IFM low-level interface between SUT and test system. The timed traces – i. e. sequences $\langle (e_1, u_1), (e_2, u_2), \ldots \rangle$ of I/O events e in abstract representation and associated time stamps – are relayed to abstract machines acting as checkers as described in Section IV.

Communication Control Layer (CCL). The *communication control layer (CCL)* relays events between abstract machines and interface modules. Observe that events generated by one abstract machine may be of interest to several recipients, so that the CCL should support some kind of multicast mechanism.

Example. Specific functions – such as switching off the cabin illumination completely – may only be performed if the aircraft is on ground. This is indicated by a discrete sensor value

```
channel ldg_down_compressed : Bool
```

which shows whether the landing gears are in state *down* and *compressed* (i. e. the weight of the aircraft can be sensed by the landing gear control unit). A typical robustness test consists in trying to trigger such a function while not on ground. To this end, a test generator simulating the landing gear control unit changes the state to

```
channel ldg_down_compressed.false
```

This state change is not only mapped by an IFM to the corresponding hardware interface of the SUT, but also relayed by the CCL to the abstract machines used to check whether execution of the associated system functions will now be blocked by the SUT.

Test Data Exchanged Between Abstract Machines and Interface Modules. Just as in specification languages used for development purposes, the data items exchanged between system under test, interface modules and abstract machines can be classified as *events* and *state information*. Events represent the discrete points in time where information is exchanged between senders and receivers. The event may be atomic (i. e. unstructured information identified by the event name) or associated with data. Due to the *finite variability* assumption which is well justified when dealing with concrete, finite-speed controllers and peripherals, only a bounded number of events can be generated and consumed in a given time interval. As long as only discrete data is processed by the SUT and its environment, events suffice to model each possible type of operations on data: They mark the points in time when interfaces or variables change their state to new discrete values.

In the context of the test system architecture described here, events may be classified as

- *Atomic events* that are in one-to-one correspondence with concrete discrete data passed along specific interfaces,
- *High-level* events which are abstractions of several atomic events,
- *Auxiliary* events which are used to
  - Coordinate different abstract machines during the test,
  - Generate requirements tracing tags which are used to relate specific test situations to associated requirements specifications,
  - Generate debugging information indicating specific states reached by abstract machines during test executions.

If *time-continuous data* (e. g., speed, thrust, temperature) is processed by the SUT or its environment, it is helpful for both system design and test design to describe this in a *hybrid system model* where the change of real-valued variables may be described by continuous (often piecewise differentiable) functions over time.

It is the task of the communication control layer to provide access for abstract machines and interface modules to both discrete events and time-continuous variables.

IV. Automated Test Evaluation

A. Overview

In this Section, a test evaluation technique is presented which offers the following features:

- Correctness criteria may be represented by
  - A specification of the SUT itself, or
  - A collection of assertions describing various behavioral SUT properties.
- The following formalisms can be used to specify correctness criteria:
  - TCSP for all correctness aspects related to the transformation of discrete data, sequencing, synchronization and timing of events,
  - Numerical conditions (e. g. differential equations or relations between R^n-valued functions over time) about time-continuous changes of real-valued variables. The conditions must be associated with numerical function libraries for evaluation.
- The correctness criteria can be allocated on a network of abstract machines acting as checkers.
- The evaluation technique is suitable for on-the-fly evaluation in soft real-time.
- The on-the-fly evaluation may be performed in hard real-time, if the following conditions are fulfilled:
  - Either the SUT specification is deterministic or only deterministic assertions are evaluated.
  - All algorithms used to evaluate time-continuous properties execute in hard real-time.

The technique is based on the results achieved by Oliver Meyer [15] and previous work on reactive systems testing without timing requirements by the author in collaboration with Michael Siegel [17], [19], [20], [21].

We focus on the checking algorithm for timed traces against model based expected results specifications described by TCSP processes P. The algorithm operates by first lifting the real-time checker process P to an untimed abstraction $\mathcal{A}(P)$ augmented by auxiliary events representing the beginning and ending of specific time intervals that are evaluated by P. This transformation is performed before starting the test execution. The timed trace s observed during the test execution is lifted to an untimed trace over the alphabet of $\mathcal{A}(P)$ using an abstraction $\mathcal{T}(s, \mathcal{A}(P), (t_1, \ldots, t_n))$ which takes the timed trace s, the untimed abstraction of P and the durations t_i of the timer intervals used by process P as arguments. The abstractions $\mathcal{A}$ and $\mathcal{T}$ are constructed so that s is a timed trace of P if and only if $s' = \mathcal{T}(s, \mathcal{A}(P), (t_1, \ldots, t_n))$ is an untimed trace of $\mathcal{A}(P)$. Construction of s' and the check whether s' is included in the alphabet of $\mathcal{A}(P)$ can be performed on the fly. To describe this concept in more detail, Section IV-B recalls results about trace checking against processes without timing requirements. Section IV-C introduces a transformation for TCSP processes P into semantically equivalent ones possessing a structure which allows us to define the abstraction mapping $\mathcal{A}(P)$ in a convenient way. Section IV-D introduces the on-the-fly checking algorithm against deterministic TCSP processes. This version operates in hard real-time and can be used to check against deterministic SUT specifications or processes encoding assertions (which are always deterministic since the evaluation result has to be deterministic). The extension for checking against nondeterministic SUT specifications is described in Section IV-E.

B. Checking Test Traces in Absence of Timing Requirements

Suppose we are performing tests where timing can be neglected. Then every test execution can be represented by the trace $u = \langle e_1, \ldots, e_k \rangle$ of the atomic, high-level or auxiliary events e_i observed in this order during the test execution, forgetting about the points in time when the e_i occurred. Since we are dealing with discrete interfaces, it is legal to assume that each interface can only transmit a finite set of data values. Therefore, the test execution u may be regarded as a word of some language $\mathcal{L}$ over the finite alphabet Σ containing all possible events e which may be observed on some interface visible in the test configuration.

Now let us assume that $\mathcal{L}$ contains exactly the words corresponding to legal test executions in the given configuration. The SUT and all abstract machines used in the test configuration can be semantically represented as finite-state labeled transition systems (LTS) [9, pp. 66] whose parallel composition is again a finite-state LTS. As a consequence, the language $\mathcal{L}$ is regular and can therefore be checked by a deterministic finite automaton [14]. As a consequence, any assertion about test execution u we might be interested in can be checked using a deterministic finite automaton. If we prefer to use the SUT specification itself as a checker and this is nondeterministic, it can – at least theoretically – be transformed into an equivalent deterministic checker.

Example. Suppose a component of the Airbus smoke detection facility SDF implements the task to react on a smoke alarm (represented by input `smk_alarm.on`) by switching on a smoke warning light in the cockpit (represented by output `smk_warn_cpt.on`) and sending a message to the flight warning system FWS (represented by output `smk_warn_fws.on`). Then the order in which the messages are delivered is arbitrary, so the specification in CSP might look like

```
SDF = (smk_alarm.on ->
          smk_warn_cpt.on ->
          smk_warn_fws.on -> SDF')
       []
      (smk_alarm.on ->
          smk_warn_fws.on ->
          smk_warn_cpt.on -> SDF')
```

Then SDF is a nondeterministic process, since in its initial state SDF, the input `smk_alarm.on` may lead to two different transitions as indicated by the external choice operator `[]`. An associated deterministic checker can be structured like

```
SDFCHK =
  smk_alarm.on ->
     ((smk_warn_cpt.on ->
          smk_warn_fws.on -> SDFCHK')
       []
      (smk_warn_fws.on ->
          smk_warn_cpt.on -> SDFCHK'))
```

During a test execution, the abstract machine running SDFCHK receives the restriction of u to events `smk_alarm.on`, `smk_warn_cpt.on` and `smk_warn_fws.on`. This subtrace of u is only accepted by the checker if it is a trace of process SDFCHK.

C. A Structural Decomposition Theorem for TCSP

The definition of Timed CSP shows that basically one operator suffices to specify real-time conditions about discrete events: The timeout expression `P [t> Q` denotes a system which acts as P if this process can be activated (by receiving an input or being able to produce an output) before the time interval of duration t elapses. If no initial event of P happens before timeout t, the system will behave like Q. The `WAIT t` operator (wait until t elapses, then skip) is just an abbreviation for `STOP [t> SKIP`.

Example. Let us construct a more complex real-time version SDF2 of the process SDF introduced above, which only reacts on a smoke alarm if it remains stable for t time units:

```
SDF2 = (smk_alarm.on ->
          (smk_alarm.off -> SDF2
            [t> (smk_warn_cpt.on ->
                  smk_warn_fws.on -> SDF2')))
       []
      (smk_alarm.on ->
          (smk_alarm.off -> SDF2
            [t> (smk_warn_fws.on ->
                   smk_warn_cpt.on -> SDF2')))
```

Application of the structural decomposition theorem proven in [15] allows us to transform SDF2 into a process SDF3 which is semantically equivalent to SDF2 in the timed failures semantics of TCSP, so that SDF3 has structure

```
SDF3 = (SDF4
         [| { setTm, elapsedTm } |]
        TIMER) \ { setTm, elapsedTm }
```

where `setTm, elapsedTm` are new auxiliary events used for synchronization between the timer management process

```
TIMER = setTm ->
          ((WAIT t; elapsedTm -> TIMER)
            []
            TIMER)
```

and SDF4. TIMER can always be triggered by a `setTm`-event. If its is not reset by another `setTm`-event within time t, it will “elapse” by generating the `elapsedTm` event. The important point of the transformation is that SDF4 does not contain any `[t>` or `WAIT t` operators:

```
SDF4 =
     (smk_alarm.on ->
       setTm ->
       ((smk_alarm.off -> SDF4)
        []
        (elapsedTm ->
           smk_warn_cpt.on ->
           smk_warn_fws.on -> SDF4')))
      []
     (smk_alarm.on ->
       setTm ->
       ((smk_alarm.off -> SDF4)
        []
        (elapsedTm ->
            smk_warn_fws.on ->
            smk_warn_cpt.on -> SDF4')))
```

Instead, SDF4 “sets a timer” by issuing `setTm` events and detects the timeout by reacting on the `elapsedTm` event.

D. An on-the-fly Checking Algorithm for TCSP – Deterministic Case

Let P be a deterministic TCSP process to be used as a checker. We define an abstraction mapping $\mathcal{A}$ from TCSP to untimed CSP as follows: $\mathcal{A}(P)$ is the untimed CSP process which syntactically equals the process component without any timeout or wait operators, as obtained from the structural decomposition theorem sketched above.

Example. For a deterministic version of SDF2 defined above, its abstraction into untimed CSP is given by process

```
SDF5 =
     (smk_alarm.on ->
       setTm ->
       ((smk_alarm.off -> SDF5)
        []
        (elapsedTm ->
          ((smk_warn_cpt.on ->
              smk_warn_fws.on -> SDF5')
            []
           (smk_warn_fws.on ->
              smk_warn_cpt.on -> SDF5'))))
```

Since the abstraction $\mathcal{A}(P)$ is also deterministic, the operator $\mathcal{A}(P)/u$ denoting the process state of $\mathcal{A}(P)$ after having run through trace u is well defined as long as u is a trace of $\mathcal{A}(P)$.

Example. The process state `SDF5/<smk_alarm.on,setTm,smk_alarm.off>` is equal to `SDF5`.

Let $\mathit{initial}(\mathcal{A}(P)/u)$ denote the set of events which may be accepted by $\mathcal{A}(P)$ after having run through valid trace u. Let $\mathit{out}(\mathit{SUT})$ denote the set of events which can be generated by the SUT on its output interfaces. The abstraction mapping for the timed trace s which is obtained during the test execution and should be checked against P is defined as follows:

```
$\mathcal{T}(s, \mathcal{A}(P), (t_1, \ldots, t_n)) =$
        $C(s, \langle\rangle, \mathcal{A}(P), (t_1, \ldots, t_n), (0, \ldots, 0), 0)$

$C(w, z, \mathcal{A}(P), (t_1, \ldots, t_n), (x_1, \ldots, x_n), x) =$
    if  $w = \langle\rangle$ then  return $z$
    else let  $(e,t) = \mathit{head}(w)$
                $\wedge\ Q = \mathcal{A}(P)/z$
                $\wedge\ S = \{i \mid \mathit{setTm}_i \in \mathit{initial}(Q)\}$
                $\wedge\ E = \{i \mid x_i \in \mathit{min}\{x_j \mid \mathit{elapsedTm}_j \in \mathit{initial}(Q)\}\}$ within
    (if  $S \neq \varnothing$ then
        $C(w, z \frown \langle \mathit{setTm}_{\mathit{min}(S)} \rangle, \mathcal{A}(P),$
              $(t_1, \ldots, t_n), (x_1, \ldots, t_{\mathit{min}(S)}, \ldots, x_n), x)$
     else if  $E \neq \varnothing \wedge x + x_{\mathit{min}(E)} \leq t$
     then $C(w, z \frown \langle \mathit{elapsedTm}_{\mathit{min}(E)} \rangle, \mathcal{A}(P),$
              $(t_1, \ldots, t_n),$
              $(\mathit{max}(0, x_1 - x_{\mathit{min}(E)}), \ldots, \mathit{max}(0, x_n - x_{\mathit{min}(E)})),$
              $x + x_{\mathit{min}(E)})$
     else if  $e \in \mathit{initial}(Q) \setminus \mathit{out}(\mathit{SUT})$
             $\wedge\ (E = \varnothing \vee t - x < x_{\mathit{min}(E)})$
     then $C(\mathit{tail}(w), z \frown \langle e \rangle, \mathcal{A}(P),$
              $(t_1, \ldots, t_n),$
              $(\mathit{max}(0, x_1 - t + x), \ldots, \mathit{max}(0, x_n - t + x)), t)$
     else if  $e \in \mathit{initial}(Q) \cap \mathit{out}(\mathit{SUT}) \wedge t = x$
     then $C(\mathit{tail}(w), z \frown \langle e \rangle, \mathcal{A}(P), (t_1, \ldots, t_n), (x_1, \ldots, x_n), x)$
     else  return error)
```

$\mathcal{T}$ is defined by using a recursive function C which returns an untimed trace of $\mathcal{A}(P)$ if the timed trace was successfully checked, otherwise **error**. The parameters of C are defined as follows: w is the rest of the timed test execution trace which still remains to be checked. Parameter z is the untimed trace constructed so far; z is a member of the trace space of $\mathcal{A}(P)$. As third parameter, C inputs the untimed abstraction process $\mathcal{A}(P)$. Next parameter is the constant vector $(t_1, \ldots, t_n)$ of positive timer values, so that timer i is always set to duration t_i (timers of variable duration could also be handled, but this would require a considerably longer specification of the abstraction mapping $\mathcal{T}$). The variable vector $(x_1, \ldots, x_n)$ stores the remaining time for each timer i until it could produce its $\mathit{elapsedTm}_i$ event. Last parameter x stores the time since start of the test execution.

C is evaluated as follows: If w is empty, this means that the whole test execution trace has been successfully checked, the associated untimed abstraction trace is contained in z and is known to be a trace of $\mathcal{A}(P)$. For non-empty w, the following abbreviations are introduced: The pair (e,t) denotes event e and timestamp t which is the next to be processed from the remaining test execution trace w. Q denotes the process $\mathcal{A}(P)$ after having run through trace z. S is the set of timer indexes whose associated *setTm*-events are accepted by $\mathcal{A}(P)$ in its actual processing state Q. E is the set of timer indexes whose associated *elapsedTm*-events have minimum time to elapse among those timers that might elapse in state Q. If S is non-empty, the *setTm*-event of the smallest index $\mathit{min}(S)$ from S is appended to the untimed trace z, and the timer value is set to its initial value $t_{\mathit{min}(S)}$. If no *setTm*-events are there to process, the *elapsedTm*-events accepted in the present process state Q are analyzed: If E is non-empty, the timer with the smallest index which is next to elapse and accepted by Q is evaluated: If the time t associated with the head-event of timed trace w indicates that (e,t) occurred after the timer with index $\mathit{min}(E)$ elapsed, the associated $\mathit{elapsedTm}_{\mathit{min}(E)}$-event is appended to the untimed trace z, the actual time value x is increased by the elapsed timer duration $x_{\mathit{min}(E)}$, and the durations of all active timers will be reduced by the same amount $x_{\mathit{min}(E)}$. Intuitively speaking, addition of this auxiliary event to untimed trace z indicates that between the previous event and $\mathit{elapsedTm}_{\mathit{min}(E)}$ "nothing has happened". The next alternative is to investigate events e processed by the checker which are not outputs of the SUT. If such an e is a member of $\mathit{initial}(Q)$, it will be always accepted, as long as we stay in state Q. If e happened before the shortest active timer duration elapses in state Q, it will be added (without its timestamp t) to untimed trace z and the timer values as well as the actual time x are accordingly adjusted. Finally, an SUT output e which is accepted in state Q must occur immediately – otherwise e would be preceded by an *elapsedTm*-event. All other pairs (e,t) result in an error: Either e is not accepted by Q but occurred before an elapsed timer triggered a state transition to an after-state of Q, or it is an SUT output expected in state Q and occurred too late.

Note that this algorithm is suitable for on-the-fly test evaluation in hard real-time: All sets involved are bounded by the constant number of timers which are used in P or by the number of transitions emanating from Q, which is bounded by the cardinality of Q's alphabet since it is deterministic. Only the head of the timed trace w has to be analyzed, so this is independent on the length of w or z. $\mathcal{A}(P)$ can be represented as the graph of a labeled transition system. Therefore finding the post-state of Q after the selected event is a one-step operation, if suitable graph encodings have been chosen.

Observe that this algorithm still has to be augmented by additional timing events to detect situations when the SUT crashes after having performed a legal sequence of events so far. Moreover, small timing deviations will be tolerated instead of using the strict equation $t = x$. These details have been omitted here due to the usual space limitations.

Example. Suppose a test execution for the SDF example introduced above resulted in

```
<(smk_alarm.on,10),(smk_alarm.off,50),
 (smk_alarm.on,70),(smk_warn_fws.on,180)>
```

and the concrete duration t after which warning messages are produced is 100 time units. The checking algorithm based on $\mathcal{A}(P) = SDF5$ will detect an error at event `(smk_warn_fws.on,180)`: It is an output of the SUT but 10 time units late.

E. An on-the-fly Checking Algorithm for TCSP – Nondeterministic Case

If the abstraction $\mathcal{A}(P)$ turns out to be nondeterministic, we could apply the usual normalization procedure used to construct the deterministic finite automaton which checks the same language [14] (Schneider describes an analogous normalization algorithm for LTS with time [28]). However, it is well known that the normalization algorithm uses considerable computation time and may lead to state explosions which cannot be handled by today's computers.

Observing that we do not require the checkers to detect errors in hard real-time we advocate another approach: The algorithm presented in Section IV-D is started as described above. Whenever more than one transition into post-states $Q_1, \ldots, Q_k$ of Q can be taken, these are marked as possible states, and the next round of the algorithm is applied to each of these states. If one of the marked states cannot handle the head of the trace, it is deleted from the list, and the remaining marked states are analyzed. If the set of marked states is empty, an error has been detected. Note that this algorithm does not require back-tracking, but the number of marked states can become quite large. Observe further that moving to different possible post-states of Q results in different untimed traces z, but fortunately, the z-traces are not really needed for a practical test evaluation.

V. Conclusion

We have presented a list of problems to be solved for automated testing of hard real-time systems. Existing solutions have been sketched; a generic architecture for test automation systems has been proposed, and we have described a test evaluation algorithm which is suitable for on-the-fly evaluation in real-time.

Currently, the underlying theory is enhanced so that the full range of tests involving time-continuous variables can be handled in a well-founded manner. Initial results in this direction have been achieved by Peter Amthor in [1].

The on-the-fly checking algorithm introduced in Section IV has been implemented for abstract machines acting as checkers in the test automation system RT-Tester [26]. Tool qualification according to the regulations of [8] has been achieved for RT-Tester with respect to test of the CIDS controller in the A340-500/600 and A318 aircrafts and for testing the A318 smoke detection facility. Further qualification suites for other controllers are currently prepared.

Acknowledgments. A major part of the theoretical results investigated by the author and his research team at the University of Bremen and described in this article has been elaborated within the HYBRIS project (Efficient Specification of Hybrid Systems) supported by the Deutsche Forschungsgemeinschaft DFG as part of the priority programme on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* [7]. The application-oriented research for testing avionics control systems is carried out within the European VICTORIA research project focusing on new architectural, development and validation concepts integrating on-board functions for commercial airliners. The author would like to express his thanks to the members of the IDPT 2002 conference committee for their invitation to present this article.

References

[1] Amthor, Peter (2000). *Structural Decomposition of Hybrid Systems.* Dissertation, Universität Bremen, Mar. 2000. Published in Gogolla *et al.* (Eds.): Monographs of the Bremen Institute of Safe Systems, Volume 13.

[2] Apt, Krzysztof R. and Olderog, Ernst-Rüdiger (1991). *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlin Heidelberg New York.

[3] Bredereke, Jan and Schlingloff, Bernd-Holger (2001). Specification Based Testing of the UMTS Protocol Stack. In *Proc. of the 14th Int'l. Software&Internet Quality Week - QW2001, San Francisco, USA (29 May - 1 June 2001).* Available on CD-ROM.

[4] Bredereke, Jan and Schlingloff, Bernd-Holger (2002). An Automated, Flexible Testing Environment for UMTS. In *Proceedings of the 3rd ICSTEST, International Conference on Software Testing, Düsseldorf, April 17th – 19th.*

[5] Brinksma, Ed and Tretmans, Jan. (2000) Testing transition systems: An annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Proceedings of Summer School MOVEP'2k Modelling and Verification of Parallel Processes, Nantes, July 2000.* pp. 44-50.

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

[7] Deutsche Forschungsgemeinschaft DFG. Priority Programme *Software Specification – Integration of Software Specification Techniques for Applications in Engineering.*
Information available under
`http://tfs.cs.tu-berlin.de/projekte/indspec/SPP`

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

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

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

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

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

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

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

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

[16] 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.

[17] 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.

[18] Peleska, Jan (1996). Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. 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.

[19] 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.

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

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

[22] 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.

[23] Peleska, Jan (2002). Hardware/Software Integration Testing for the new Airbus Aircraft Families. In *Proceedings of the TestCom 2002 Testing Internet Technologies and Services, Berlin, March, 19th-22nd 2002.*

[24] Peleska, Jan and Tsiolakis, Aliki (2002). Automated Integration Testing for Avionics Systems. In *Proceedings of the 3rd ICSTEST, International Conference on Software Testing, Düsseldorf, April 17th – 19th.*

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

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

[27] 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.

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

[29] Schronen, Michael (1999). *Methodology for the Development of Microprocessor-Based Safety-Critical Systems.* Dissertation, Universität Bremen, Sep. 1998 Published in Gogolla *et al.* (Eds.): Monographs of the Bremen Institute of Safe Systems, Volume 8. Shaker Verlag, ISBN 3-8265-4521-4.

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