Model-Based Testing for the Second Generation of Integrated Modular Avionics

Christof Efkemann, Jan Peleska
Department of Computer Science and Mathematics
University of Bremen
Bremen, Germany
Email: {chref, jp}@tzi.de

***Abstract*—In this paper the authors present the current research and development activities regarding automated testing of Integrated Modular Avionics controllers in the European research project SCARLETT. The authors describe the goals of the SCARLETT project and explain its background of Integrated Modular Avionics. Furthermore, they explain different levels of testing of components required for certification. A domain-specific modelling language designed for the IMA platform is presented. This language is used to create models from which tests of different levels can be generated automatically. The authors expect significant improvements in terms of effort to create and maintain test procedures compared to conventional test creation.**

***Keywords*-IMA; SCARLETT; TTCN-3; avionics; domain-specific modelling; model-based testing;**

I. Introduction

This section starts with an introduction to the Integrated Modular Avionics platform, its use in modern aircraft and its specific testing needs, followed by a short overview of the European research project SCARLETT and its goals. A detailed description of the authors' contributions to this project is given and we refer to related work in this field.

A. Integrated Modular Avionics

The traditional *federated* aircraft controller architecture [1, p. 11] consists of a large number of different, specialised electronics devices, many of them with custom interfaces. In the *Integrated Modular Avionics (IMA)* architecture this multitude of device types is replaced by a small number of modular, general-purpose component variants whose instances are linked by a high-speed data network. Due to high processing power each module can host several avionics functions, each of which previously required its own controller. The IMA approach has three main advantages:

- Reduction of weight through a smaller number of physical components and reduced wiring, thereby increasing fuel efficiency.
- Lower maintenance costs by reducing the number of different types of replacement units needed to keep on stock.
- Reduction of development costs by provision of a standardised operating system, together with standardised drivers for the avionics interfaces most widely used.

The work presented in this document has been produced with funding received from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement no. ACP7-GA-2008-211439.

An important aspect of module design is *segregation*: In order to host applications of different safety assurance levels on the same module, it must be ensured that those applications cannot interfere with each other. Therefore a module must support resource partitioning via memory access protection, strict deterministic scheduling and I/O access permissions. Bandwidth limitations on the data network have to be enforced as well. The interface between applications and the operating system conforms to a standardised API which is specified in the ARINC 653 standard [1]. This standard also defines the scheduling properties ensuring partitioning in the time domain.

Due to their modularity and flexibility IMA components have a complex configuration. It covers many aspects like partitioning, resource allocation, scheduling, I/O configuration, network configuration and internal health monitoring. This makes the configuration an integral part of the system, and it must be taken into consideration during verification and certification [2], [3]. As a result of these considerations, different levels of testing for IMA modules have been defined in a previous research project [4]:

- *Bare Module Tests* are designed to test the module and its operating system at API level. The test cases check for correct behaviour of API calls, while robustness test cases try to violate segregation rules and check that these violation efforts do not succeed. Bare module tests are executed with specialised module configurations designed for these application-independent test objectives. The application layer is substituted by test agents that perform the stimulations on the operating system as required by the testing environment and relay API output data to the testing environment for further inspection.
- *Configured Module Tests* do not focus on the module and its operating system, but are designed to test application-specific configurations meant to be used in the actual aircraft. The test cases check that configured I/O interfaces are usable as defined. Again, the application layer is replaced by test agents.

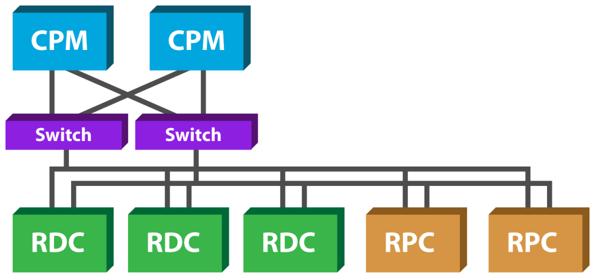

Figure 1. DME Architecture

- *Functional Tests* run with the "true" application layer integrated in the module and check for the behavioural correctness of applications as integrated in the IMA module.

The standard aircraft documentation reference for IMA is ATA chapter 42. The IMA architecture is currently in use in the Airbus A380, A400M, the future A350 XWB, and Boeing 787 Dreamliner aircraft.

B. SCARLETT

SCARLETT, meaning "SCAlable & ReconfigurabLe elEctronics plaTforms and Tools", is a European research and technology project [5]. With 38 partner organisations (airframers, large industrial companies, SMEs and universities) it is a large-scale integrating project funded under the Seventh Framework Programme [6] of the European Commission. SCARLETT is a successor to several earlier European research projects in the field of avionics, like PAMELA, NEVADA and VICTORIA. Its goal is the development of a new generation of IMA with increased scalability, adaptability and fault-tolerance, called *Distributed Modular Electronics (DME)*.

The DME concept aims at the separation of processing power from sensor/actuator interfaces, thereby reducing the number of different component types to a minimum. This also makes DME suitable for a wider range of aircraft types by giving system designers the possibility to scale the platform according to required hardware interfaces and computing power. Figure 1 shows an example of a network of components: two Core Processing Modules (CPM), three Remote Data Concentrators (RDC) and two Remote Power Controllers (RPC) linked via two redundant AFDX[1] networks. The CPM components provide the computing power and host the avionics applications, while the RDC and RPC components provide the required number of sensor/actuator and bus interfaces.

The project also investigates ways of increasing fault tolerance through different reconfiguration capabilities, e. g. transferring avionics functions from defective modules to other operative modules. Finally, the design of a unified tool chain and development environment will lead to improvements of the avionics software implementation process.

The project is currently in its implementation phase. It will finish in October 2011. The authors are involved in the tool development aspects of the project. Their goal is to provide tools for automated model-based testing of DME components and applications. The automation scope covers test case, test data and test procedure generation from domain-specific models as well as the test execution in a hardware-in-the-loop testing environment.

C. Main Contributions

In section II the authors introduce a domain-specific modelling language (DSL) for IMA configuration and IMA behaviour. This language allows test designers to specify test cases for bare and configured module tests in a very compact manner, without explicitly referring to the availability and distribution of resources[2] in the IMA module under test. A generator produces the concrete test procedure in TTCN-3 syntax. This procedure stimulates and controls distributed interactions between test agents on different partitions and modules. For this purpose, the generator evaluates both the test case specification and the IMA configuration (which is also generated automatically in the case of bare module tests) and selects the concrete resources involved in compliance with the configuration. For HW/SW integration tests with the application layer present in each IMA module which is part of the system under test (SUT), behavioural test models based on timed state machines can be used, and the generator derives test cases, test data and procedures from these models by means of a constraint solver.

The effectiveness of our solution is assessed by comparing the effort needed for test model development in the new approach described here to the effort necessary to write TTCN-3 procedures covering the same test objectives in a manual way.

D. Related Work

The results related to bare and configured module tests described in this paper are based on earlier work completed within the VICTORIA project [4], [9]. In that project a library consisting of formal Timed CSP [10], [11] specification templates had been developed to facilitate the creation of bare module tests. The amount of manual effort for test procedure creation, however, was still substantial. Moreover, domain experts without knowledge in the field of process algebras were reluctant to analyse the resulting test procedures. In contrast to that, the approach presented here uses an intuitive graphic template formalism to describe patterns from which concrete test cases and test data can be derived

[1]The *Avionics Full DupleX (AFDX) Switched Ethernet* network is defined in the ARINC 664 standard [7] and used as high-speed communication link between aircraft controllers. It is the successor of the slower ARINC 429 networks [8] and used, for example, in Airbus A380 and A350XWB aircraft as well as in the Boeing 787 Dreamliner.

[2]Examples for resources are partitions, threads, communication ports, semaphores and blackboards for shared memory data exchange.

and which are automatically "wrapped" in executable test procedures. Additionally, the VICTORIA results relied on specific test engine hardware and test execution software. In contrast to that, the SCARLETT project tries to achieve independence from specific test hardware by deploying a TTCN-3-based test environment [12]–[14]. The underlying techniques for model-based functional testing are described in more detail in [15], where also references to competing approaches are given.

Domain-specific approaches to test automation are also promising in other areas: In [16], for example, patterns for automated test generation in the railway control system domain are described.

II. Domain-specific Modelling Language ITML

In this section the authors present the domain-specific language components for bare module, configured module and functional HW/SW integration tests and illustrate how concrete test cases and test data, as well as the test procedures executing these test cases are generated automatically from the DSL test case specification templates. Section IV contains the conclusion and gives indications with respect to future work.

A. Domain-specific Modelling in the IMA Testing Domain

In general, domain-specific modelling (DSM) makes use of graphical domain-specific languages (DSL) consisting of elements from a certain problem domain. These models can easily be understood, validated and even created by domain experts without requiring in-depth knowledge of software engineering principles [17]. For the first generation of IMA, test procedures were implemented manually. This proved to be time-consuming and error-prone. Furthermore, the domain experts could not write the test procedures by themselves – only people with profound knowledge of the test environment were able to do that. The authors' intention is to improve this situation for the second generation of IMA. Therefore, they designed the IMA Test Modelling Language (ITML), a language for the IMA testing domain.

As described in the introduction, testing of IMA modules is performed on different levels. This is reflected in ITML through different variants of the language: ITML-B is designed for bare module testing, while ITML-C is suited for configured module testing. A third variant, ITML-A, designed for application testing, will be introduced further below.

The metamodels and models presented here have been implemented using the DSM tool MetaEdit+ [18].

B. Bare Module Tests: ITML-B

The IMA Test Modelling Language variant B allows the definition of bare module tests. Since there are no predefined configurations for bare module testing, the language allows the definition of suitable configurations, in addition to API-oriented test actions. Therefore, ITML-B is split into two parts: the configuration modelling part allows the definition of a set of configurations for a test case, while the behaviour modelling part allows the specification of the dynamic test case behaviours.

1) Configuration: The goal of the configuration part is not to provide a full-featured configuration editor (which would be very tedious to use, due to the large number of configuration parameters available), but to allow the user to describe constraints on a set of configurations with a minimum of effort. ITML-B allows the test designer to specify the components relevant for his test case, like partitions, processes and ports, in a "plug-and-play" manner. However, for each attribute, the designer can specify a single value, or a range of values. The latter results in not one, but a set of configurations being generated from the specification.

In detail, the metamodel defines a three-level tree of objects: Its root is a *CPM Config* object. The next level consists of *Partition Config* objects, followed by *Process Config* objects. The metamodel provides a relationship that models the edges of the tree. A *Process Config* object describes one or more processes and is linked to a *Partition Config* object. The *Partition Config* object describes one or more partitions, and each of those partitions contains processes as described by the linked *Process Config* object(s). The same holds for *CPM Config* and *Partition Config*: The CPM described by the *CPM Config* object contains partitions as specified by the associated *Partition Config* objects.

Communication ports between partitions and to other remote components are not specified one by one, but instead as sets of ports of a specified type. This level of abstraction helps test designers to quickly generate suitable configurations for their test cases. For that purpose the metamodel provides the objects *AFDX Port Config* and *RAM Port Config*. A second type of relationship is used here to link the port groups to partitions and to denote the direction of the dataflow between partitions and ports.

A great advantage of modelling is the use of constraints in the metamodel. The modelling tool always ensures the compliance with those constraints, thereby guaranteeing that the relations in configuration models are correct and that the models are valid.

Figure 2 shows an example of a configuration model.

A generator written in MERL[3] exports the configuration models to files in an intermediate XML format. These files are then converted into actual configurations that can be processed by the module tool chain and loaded onto a CPM. In this step the generator also takes care of handling ranges of configuration values by generating multiple configurations. The generator tries to keep the overall number of generated configurations to a minimum by using different values of each ranged attribute in each configuration instead

[3]MetaEdit+ Reporting Language

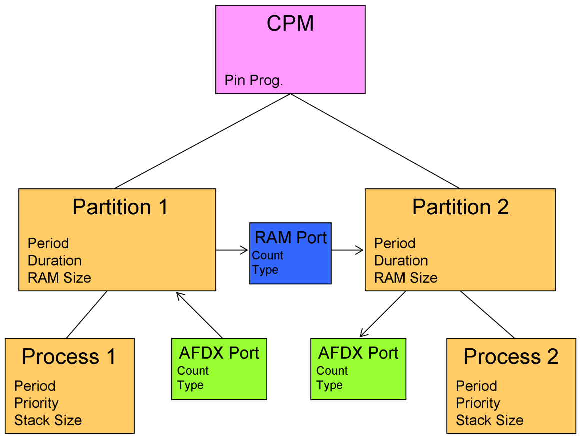

Figure 2. ITML-B Configuration example

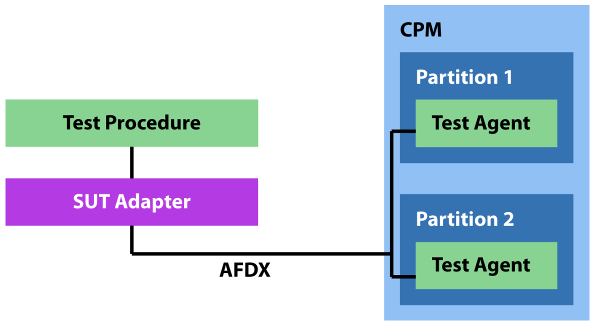

Figure 3. Test Agent

of producing a Cartesian product.

2) Behaviour: In order to understand the ideas behind the modelling of the behaviour part it is important to recall that there is no real application loaded onto the module for bare module and configured module testing. A generic test agent (TA) is used instead. An instance of the TA is loaded into each partition of the module (see Figure 3). The test procedure uses pre-defined AFDX ports to communicate with the TAs on the module and have them perform API calls on its behalf. A remote procedure call protocol is used to encode functions, parameters and return values.

An ITML-B behaviour part specifies a generic test case that is to be executed with one or more configurations. The behaviour part basically resembles a flowchart.

The metamodel describing valid behaviour models is more complex than the configuration metamodel described before. It contains objects of different categories: *Partition* and *Process* objects serve as containers for other objects, defining their context. *Start*, *Pass* and *Fail* are flow control objects and specify where the flow starts end ends. *Blackboard*, *Buffer* and *Message* are resource objects. They are used in combination with *API Call* and *Complement* objects. The *For all* object is another container, providing a loop construct.

API Call objects are placed inside a container object, thereby defining where the respective API call is to be executed. The metamodel provides a *Parameter* relationship which links resource objects to *API Call* objects. The *For all* loop container provides a default argument to the API calls contained in it. A second type of relationship provides the edges of the flowchart, linking the flow control objects with the *API Call* and *Complement* objects.

Complement objects (either “read complement” or “write complement”) allow access to the other end of the resource currently in use without having to know (and hard-code) where (i. e. which module, partition) that is actually located. The currently used AFDX port can be, for example, linked to an AFDX port in another partition, or connected to a remote component, depending on the configuration.

An example of a behaviour model is given in Figure 4. The basic DSL objects in the flowchart (*Create_Sampling_Port*, *Write_Sampling_Message*) are *API Call* objects representing ARINC 653 API calls which are controlled remotely by the test procedure running on the test bench and executed by the corresponding partition’s test agent on reception of the remote procedure call. The sequence of API calls is controlled by the flowchart arrows decorated with guard conditions. Parameters of API calls are either defaults, e. g. derived from an API port currently in use, or specified explicitly, e. g. in order to use invalid values for robustness tests. Execution starts at the flow start object and ends at one of the flow end objects, each of which denotes a test case result (pass/fail).

Message objects specify message payloads, either as concrete values or as sets of allowed values, e. g. regular expressions. In Figure 4, the ∀ *RAM Sampling Output Port* box is a loop object, binding communication ports to the API calls inside the bounding box. The flowchart step objects are enclosed in a container object (outer box in Figure 4). The process frame specifies the partitions and processes on which the steps shall be executed.

Following an XML export of the model, a generator is employed here as well for the creation of TTCN-3 test procedures from the behaviour models. The generator translates the control flow graph of the flowchart into TTCN-3 statement blocks linked by if/else and goto statements. The API calls are converted into remote procedure calls to be executed by the test agent on the module. The remote procedure calls are implemented using TTCN-3’s procedure-based communication mechanism – as opposed to other I/O, which is message-based. Thereby call statements can be used to trigger remote procedure calls. Complement objects are transformed into the appropriate operations depending on the opposite end-point of the respective current communications port.

The following code fragment[4] shows an example of a

[4]A few comments have been added to make it easier to understand.

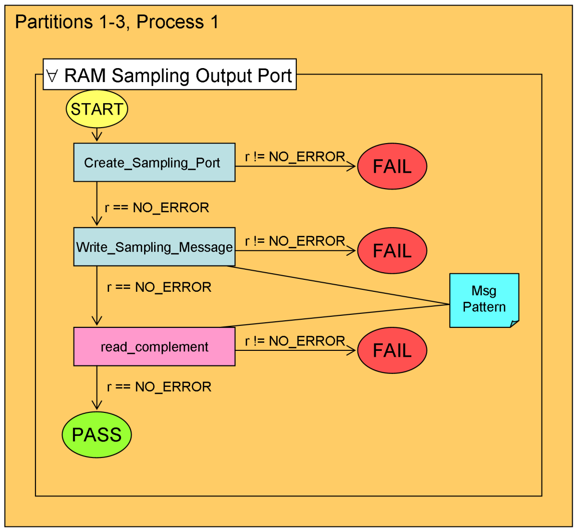

Figure 4. ITML-B behaviour example: RAM ports

generated TTCN-3 test procedure, corresponding to the model from Figure 4, to be executed on the test bench:

```
// the port ID of the current port
var PORT_ID port_id;

// first RAM sampling output port of partition 1:
label l_port1;
// first step: Create_Sampling_Port API call
ta_pt1.call(CREATE_SAMPLING_PORT:{"RAM_SP1", 128, SOURCE, 500, -}) {
  // return with NO_ERROR: save returned port ID in local variable
  [] ta_pt1.getreply(CREATE_SAMPLING_PORT:{-,-,-,-,?}
         value RETURN_CODE_TYPE:NO_ERROR) ->
         param(-,-,-,-,port_id) {}
  // other return value: FAIL and skip to next port
  [] ta_pt1.getreply(CREATE_SAMPLING_PORT:{-,-,-,-,?}) {
         setverdict(fail);
         goto l_port2;
     }
}

// second step: Write_Sampling_Message API call, use stored port ID
ta_pt1.call(WRITE_SAMPLING_MESSAGE:{port_id, msg_tmpl}) {
  // return with NO_ERROR
  [] ta_pt1.getreply(WRITE_SAMPLING_MESSAGE:{-,-}
         value RETURN_CODE_TYPE:NO_ERROR) {}
  // other return value: FAIL and skip to next port
  [] ta_pt1.getreply(WRITE_SAMPLING_MESSAGE:{-,-}) {
         setverdict(fail);
         goto l_port2;
     }
}

// third step: read_complement implemented as Read_Sampling_Message
// in partition 3, port ID determined from configuration
ta_pt3.call(READ_SAMPLING_MESSAGE:{ram_sp7_id, -, -}) {
  // return with NO_ERROR, matching message and validity
  [] ta_pt3.getreply(READ_SAMPLING_MESSAGE:{-, msg_tmpl, VALID}
         value int:NO_ERROR) {}
  // other return value, msg or validity: FAIL and skip to next port
  [] ta_pt3.getreply(READ_SAMPLING_MESSAGE:{-, ?, ?}) {
         steverdict(fail);
         goto l_port2;
     }
}
setverdict(pass);

label l_port2;
ta_pt1.call(CREATE_SAMPLING_PORT:{"RAM_SP2", 64, SOURCE, 1000, -}) {
  [] ta_pt1.getreply(CREATE_SAMPLING_PORT:{-,-,-,-,?}
         value RETURN_CODE_TYPE:NO_ERROR) ->
         param(-,-,-,-,port_id) {}
  [] ta_pt1.getreply(CREATE_SAMPLING_PORT:{-,-,-,-,?}) {
         setverdict(fail);
         goto l_end;
     }
}

ta_pt1.call(WRITE_SAMPLING_MESSAGE:{port_id, msg_tmpl}) {
  [] ta_pt1.getreply(WRITE_SAMPLING_MESSAGE:{-,-}
         value RETURN_CODE_TYPE:NO_ERROR) {}
  [] ta_pt1.getreply(WRITE_SAMPLING_MESSAGE:{-,-}) {
         setverdict(fail);
         goto l_end;
     }
}

ta_pt2.call(READ_SAMPLING_MESSAGE:{ram_sp3_id, -, -}) {
  [] ta_pt2.getreply(READ_SAMPLING_MESSAGE:{-, msg_tmpl, VALID}
         value int:NO_ERROR) {}
  [] ta_pt2.getreply(READ_SAMPLING_MESSAGE:{-, ?, ?}) {
         steverdict(fail);
         goto l_end;
     }
}
setverdict(pass);

label l_end;
```

Note how a relatively simple and small model is used to generate a much larger amount of code for the test procedure. Even if more RAM ports were defined in the configuration, the behaviour model would remain the same, but the generator would automatically create additional code in the test procedure. You can also see how the *read_complement* steps generate remote procedure calls into other partitions (`ta_pt1`: command port to TA in partition 1, `ta_pt2`: command port to TA in partition 2, etc.)

C. Configured Module Tests: ITML-C

The IMA Test Modelling Language variant C allows the definition of configured module tests. Since configurations are pre-defined in this scenario, the language does not admit the definition of configurations. Therefore, ITML-C consists only of a behaviour part. Its metamodel is identical to the ITML-B behaviour specification. This also means that ITML-B test case models can be re-used as ITML-C test cases.

The configurations (which are part of the SUT in configured module testing) are used together with the behaviour models as inputs in the generation process. The resulting TTCN-3 test procedures are tailored to the specific configuration. Whenever the module integrator releases a new version of the configuration, the tester only has to re-generate the test procedures to produce a new test suite adapted to the new configuration.

D. Hardware/Software Integration Tests: ITML-A

The natural complementation of bare module and configured module tests are functional HW/SW integration tests with the integrated application layer, so ITML-B and C are complemented by the third modelling language ITML-A(pplication). As in ITML-C, functional tests work with dedicated configurations, but the SUT boundaries applicable in ITML-A differ from those of ITML-C: for functional HW/SW integration testing IMA partitions run the target applications so that no test agents are present. The SUT boundaries consist of the hardware interfaces of the IMA module, so the test equipment acts on HW interfaces – such as AFDX, CAN, discrete and analogue – only. The underlying test case generation strategy of ITML-A is to evaluate a structural and functional model of the application behaviour and derive both test stimulations and test oracles (checkers for expected results) from this model.

To this end, the ITML-A metamodel provides a hierarchy of graphs. The top level consists of a *System Diagram* which

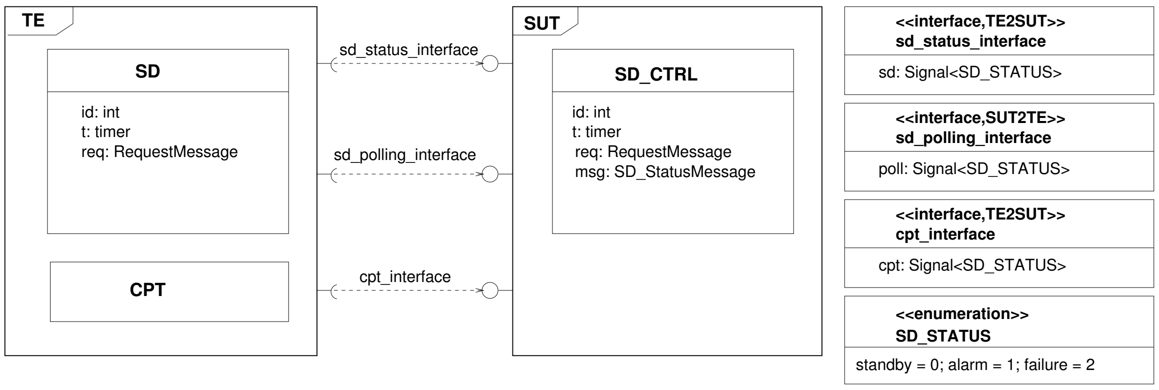

Figure 5. ITML-A state machine: composite structure diagram

is a variant of UML2.0 composite structure diagrams. It contains the SUT object and a TE (Test Environment) object; both may be decomposed further into sub-components. The system diagram also contains a list of the input and output signals between the SUT and the TE. Behaviour is specified in the leaves of the system decomposition by means of *timed state machines*.

Figure 5 shows a system diagram modelling a simplified smoke detection function: The SUT consists of an IMA computer hosting the *smoke detection control function* SD_CTRL. This function polls smoke detectors (SD) in 2-second cycles and reports smoke alarms received from SDs to the cockpit (CPT). If SDs report failures or stop answering the polling requests in time their failure is also reported to CPT.

Behaviour is modelled by timed state machines, as shown in Fig. 6 for the SD control function. In addition to local and shared variable manipulations these state machines admit timers and communication events. Variable t is a timer which acts like a stop watch. It can be set to zero using the t.reset() operation and questioned whether a time interval of length c has elapsed since the last reset, using the t.elapsed(c) operation. Communications consist of put(msg)-actions (send a message) and get(msg)-events (wait for reception of message). They act on *signals* which are automatically mapped to observable actions and events on the interfaces shared between IMA and test equipment. For SUT components, these state machines are deterministic: in case of several guard conditions being simultaneously enabled for transition emanating from a control state, the one with the highest priority is taken (0 = best priority). This can be seen in control state WAIT_RESPONSE, where the transition to state FAILURE take precedence over a transition to EVALUATE, if both guard conditions are true. Parallel components act *synchronously*, that is, they all evaluate the same pre-state and perform their associated state transitions with accompanying actions in a simultaneous way in zero time. Time passes when all components are in a stable state, and after such a positive delay new values may be placed on the inputs to the SUT. Multiple instances of components are specified using a tabular notation (not shown here due to space limitations). In our example, separate copies of SD_CTRL run concurrently in the SUT, one for each smoke detector connected to the SUT. Each instance is identified by their unique id-value which is also used to identify the signals to be used in communications with SDs and CPT.

The specification of (a part of) the test environment is optional. If no TE model is given, the test case generator will derive test cases and associated test data from the SUT model alone. To this end, the generator first identifies *symbolic test cases* as reachability goals in the model, taking into account the dependencies between concurrent components. In the simple example presented here, no dependencies between different instances of SD_CTRL are present, so the generator interleaves the goals associated with each instance in a random manner. Different model coverage goals can be configured and used in combination, such as, for example MC/DC model coverage (see [19] for a comprehensive discussion of model coverage metrics). Consider, for example, the goal to cover the transition from location WAIT_RESPONSE to FAILURE. The generator transforms this goal into a logical constraint which looks like

$$\mathsf{WAIT\_RESPONSE} \wedge (\hat{t} < t + 0.5)$$

Here the first conjunct is a Boolean variable indicating whether the SUT is in location WAIT_RESPONSE, and $(\hat{t} < t + 0.5)$ is an internal resolution of the t.elapsed(0.5) condition: the generator handles timers as ordinary variables, where the actual model execution time $\hat{t}$ is assigned to at every t.reset(). An elapsed condition can therefore be expressed as an evaluation whether the current time is greater or equal to the last time the timer was reset plus the wait time specified in the elapsed parameter. The goal constraint is combined with another constraint specifying the SUT transition relation. This is used by an SMT constraint

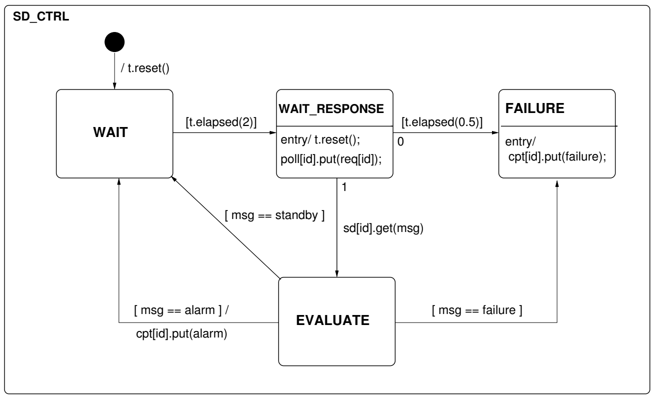

Figure 6. ITML-A state machine: component behaviour of smoke detection control function.

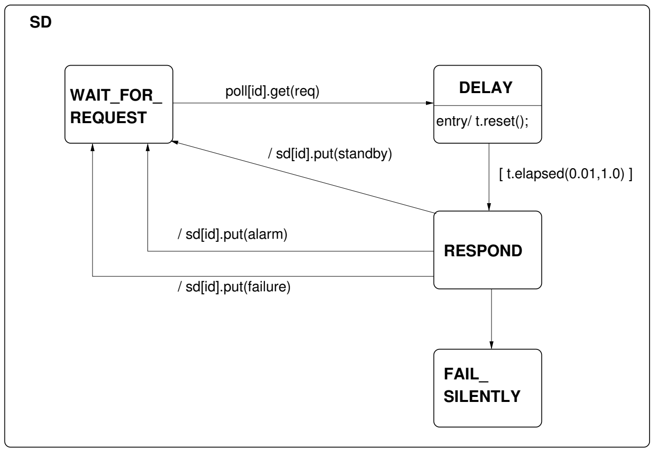

Figure 7. ITML-A state machine: TE simulation for SD behaviour

solver [20] to calculate concrete input values and time stamps at which these values are passed to the SUT, such that the goal condition is fulfilled after a finite number of system transitions. In the example, this would lead to a concrete test case (1) Wait 2 time units, (2) check for polling message to arrive, (3) Wait for a value of $0.5 + \varepsilon$ time units, (4) check for failure message to arrive at CPT.

If TE model components are present, the test case generator calculates timed input sequences to the SUT as sketched above, but now only sequences are produced which are consistent with the specified TE components. Take, for example, the TE simulation of a smoke detector as given in Fig. 7. It is also modelled using timed state machines, but for environment simulations we admit *non-deterministic* behaviour: in control state RESPOND it may be non-deterministically decided whether to respond to a polling request with a standby, alarm or failure status, or whether not to respond at all. Moreover, non-determinism is also available in the time domain. Condition t.elapsed(0.1,1.0) guarding the transition from DELAY to RESPOND becomes true after an arbitrary amount of time $\delta \in [0.01, 1.0]$ has passed since the reset of t. Taking into account this TE simulation, the generator will create the same timed sequence of test steps as described above, but observe the additional constraint in step (3) that $0.5+\varepsilon$ should not exceed 1.0, since this is the maximal delay δ for a response to be given according to the SD simulation.

Observe that currently this non-determinism is resolved by the test case generator during the generation process: it chooses among several discrete transitions which are simultaneously enabled or between different delay values with the objective to (1) reach the SUT model coverage goals which are still open and (2) maximise the coverage of behaviours possible for the TE simulations.

III. Estimation of Efficiency Gain

The application of model-based testing is a completely novel approach in the IMA domain. Compared to the manual implementation of tests, however, the qualitative advantages to be expected are obvious, since the complete test procedure elaboration process is automated. Experiences gained from the analysis of conventional test campaigns performed by the authors in cooperation with Verified Systems International GmbH in Bremen and evaluation of model-based test campaigns in other domains allow to estimate the increase of efficiency when switching from conventional to model-based testing as follows. Conventional HW/SW integration test campaigns are performed along the following lines: (1) test case and test data elaboration, (2) development of re-usable test and simulation library extensions, (3) test procedure programming, (4) test procedure trial and debugging, (5) analysis of test execution results in the HW/SW integration test environment, (6) analysis of results together with the development team, (7) test documentation and test project management. For conventional test campaigns 40% of the overall effort is allocated to steps (1) — (4) and another 40% are allocated to steps (5) — (6).

Under ideal circumstances automated model-based testing will reduce the effort for steps (1) — (4) to 0, and that of steps (5) — (6) to half the effort, because model-based testing offers the possibility to replay the test execution observed against the model and to analyse discrepancies between observed and expected behaviour on model level. As a consequence, the overall effort for a HW/SW integration test campaign will be reduced in the best case to 40% of the effort required for a conventional test campaign execution, *if test models are already available*. The initial development of new test models requires an effort which is less than that needed for steps (1) — (4). As a consequence, a model-based test project where new models have to be created is expected to require less than 80% of the effort for conventional campaigns.

IV. Conclusion and Future Work

In this article the authors have given an introduction into the Integrated Modular Avionics architecture and have shown its future development in the context of the SCARLETT research project.

While employing TTCN-3 as an executable test specification language, the test cases of interest, however, are specified on a higher level by means of domain-specific models. The concrete test procedures, which can be executed on heterogeneous test engine platforms, are generated from these high-level models in an automated way. To this end, the authors have introduced the domain-specific modelling language ITML. For bare and configured module testing it gives domain experts the ability to specify tests in a way that completely abstracts from the actual test environment. For Hardware/Software Integration testing an even higher level of automation is reached because the test cases are derived from behavioural models of the applications. This approach is estimated to reduce the overall effort to 40% — 80% of the effort required for conventional testing campaigns.

To demonstrate the usability of ITML-B in particular the authors intend to model the ARINC 653 conformity test specification [21] in ITML-B and show that it is possible to derive platform-specific conformity tests in an automatic way by means of first generating platform-independent TTCN-3 conformity test procedures from the ITML-B model and then running the test suite with platform-specific adapters. The standard [21] suggests that conformity test suites should be written as C programs, to be modified each time with respect to the platform-specific parameters. In contrast to that, the authors' approach utilises re-usable test agents executing API calls remotely controlled and monitored by the test engine. The authors expect that this concept will allow to perform the specified ARINC 653 conformity tests with considerably less effort and additional error detection capabilities, since the test procedures are generated automatically from the ITML-B specification and may therefore always test a range of configuration parameters and API call parameter variants which is much wider than the range possible for manually created test applications.

References

[1] *ARINC SPECIFICATION 653P1-2: Avionics Application Software Standard Interface, Part 1 – Required Services*, Aeronautical Radio, Inc., Dec 2005.

[2] *DO-178B: Software Considerations in Airbone Systems and Equipment Certification*, RTCA, Inc., Dec 1992.

[3] *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*, RTCA, Inc., Nov 2005.

[4] A. Ott, "System testing in the avionics domain," Ph.D. dissertation, University of Bremen, Dec 2007.

[5] SCARLETT Consortium, "Project website," http://www.scarlettproject.eu/.

[6] Publications Office of the European Union, "Website of the Seventh Framework Programme," http://cordis.europa.eu/fp7/.

[7] *ARINC SPECIFICATION 664P1-1: Aircraft Data Network, Part 1, Systems Concepts and Overview*, Aeronautical Radio, Inc., Jun 2006.

[8] *ARINC SPECIFICATION 429P1-17 Mark 33: Digital Information Transfer System (DITS), Part 1, Functional Description, Electrical Interface, Label Assignments and Word Formats*, Aeronautical Radio, Inc., May 2004.

[9] A. Ott and T. Hartmann, "Domain specific V&V strategies for aircraft applications," in *6th ICSTEST International Conference on Software Testing*, Düsseldorf, Apr 2005.

[10] S. Schneider, *Concurrent and Real-time Systems – The CSP Approach*. Chichester: John Wiley & Sons, Ltd, 2000.

[11] J. Peleska, "Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family," in *Proc. of the Sixth Biennial World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, California, June 23-28, 2002*. Society for Design and Process Science, Jun. 2002, iSSN 1090-9389.

[12] *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, ETSI, Jun 2009, ETSI ES 201 873-1.

[13] *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)*, ETSI, Jun 2009, ETSI ES 201 873-5.

[14] C. Wilcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz, *An Introduction to TTCN-3*. Chichester: John Wiley & Sons, Ltd, Apr 2005, ch. 12.

[15] J. Peleska, E. Vorobev, and F. Lapschies, "Automated test case generation with SMT-solving and abstract interpretation," in *Proceedings of the Nasa Formal Methods Symposium NFM2011*, ser. LNCS, vol. 6617. Springer, 2011, to appear.

[16] K. Mewes, "Domain-specific modelling of railway control systems with integrated verification and validation," Ph.D. dissertation, University of Bremen, Mar 2010.

[17] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Hoboken, NJ: John Wiley & Sons, Inc., Apr 2008.

[18] MetaCase, "MetaEdit+ product website," http://www.metacase.com/products.html.

[19] S. Weißleder, "Test models and coverage criteria for automatic model-based test generation with uml state machines," Ph.D. dissertation, Humboldt-Universität zu Berlin, 2010.

[20] S. Ranise and C. Tinelli, "Satisfiability modulo theories," *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, vol. 21, no. 6, pp. 71–81, 2006.

[21] *ARINC SPECIFICATION 653P3: Avionics Application Software Standard Interface, Part 3 – Conformity Test Specification*, Aeronautical Radio, Inc., Oct 2006.