Demonstration of a Formal Method for Incremental Qualification of IMA Systems

Jonas Elmqvist, Simin Nadjm-Tehrani, Kristina Forsberg and Stellan Nordenbro

Jonas Elmqvist, Simin Nadjm-Tehrani, Kristina Forsberg and Stellan Nordenbro, Demonstration of a Formal Method for Incremental Qualification of IMA Systems, 2008.
http://dx.doi.org/10.1109/DASC.2008.4702860

DEMONSTRATION OF A FORMAL METHOD FOR INCREMENTAL QUALIFICATION OF IMA SYSTEMS

Jonas Elmqvist, Simin Nadjm-Tehrani, Linköping University, Linköping, Sweden
Kristina Forsberg, Stellan Nordenbro, Saab Avitronics, Jönköping, Sweden

Abstract

In this paper we address the process of incremental certification/qualification of Integrated Modular Avionic (IMA) systems. The work aims to show that given a scalable avionics architecture we can apply a component-based development process and save some safety analysis effort by reusing design models for components. This creates a formal framework for IMA system safety assessment.

Introduction

Developing a formal framework for compositional safety analysis requires a formal representation of the system and its components. We use the notion of a safety interface that is part of an earlier developed formal component model for analysis of fault tolerance. The safety interface describes how the component behaves with respect to violation of a given system level property in presence of faults in its environment. This approach is supported by proof-of-concept tools, and provides a link between formal analysis of components in safety-critical systems and modern engineering processes supported by model-based development. In this paper we demonstrate the overall approach by showing how an upgrade of an existing system can be carried out and reuse some of the analyses of the earlier generation by focusing on the safety interfaces and compositional reasoning.

The approach is demonstrated in a test environment at Saab Avitronics. The functionality used in the test environment is an Altitude component and a Voter component. These software components come with a safety interface capturing the faults that the components are resilient to. The safety interfaces are geared towards the context of the "correctness of altitude data" which is a safety-critical property at system level. The voter component exists in two different versions, a simple one and a more fault-tolerant one. Components and faults are modelled in the toolset of Scade. The impact of faults on system safety is computed using the Design Verifier in Scade that automatically propagates effects of a faulty Altitude Subsystem on aircraft safety (erroneous control).

Once the design has been analysed, the software components are integrated with a Flap Control function running on an IMA computer in the test environment. The IMA computer (incorporating a partitioning operating system) is connected to an IMA test rig which is used for demonstration control, fault emulation and data presentation. To validate the design level safety analysis in this test environment, those faults that the components were shown to be resilient to at the analysis stage are now injected at the code level. The result is verified against the flap control software in the rig

In a revision of the system the voter component is upgraded to illustrate the efficiency of the incremental analysis. This is performed by systematically connecting fault models to the new voter, thus modelling a faulty instance of the component. The purpose is to ensure that the system safety property imposed on the system is not violated after the upgrade.

Incremental Qualification

A key property of an IMA system is the modularity which opens up for the capability to add, change or upgrade functions as well as using them for different programs given that some of the certification evidence can be reused. DO 297 [1] defines the concept of incremental acceptance as:

"A process for obtaining credit toward approval and certification by accepting or finding that an IMA module, application, and/or off-aircraft IMA system complies with specific requirements.

This incremental acceptance is divided into tasks. Credit granted for individual tasks contributes to the overall certification goal. Incremental acceptance provides the ability to integrate and accept new applications and/or modules, in an IMA system, and maintain existing applications and/or modules without the need for re-acceptance."

Approval of an IMA system installation may be based on the accumulation of incremental acceptance. The incremental acceptance may be granted in the form of an acceptance letter, i.e. acknowledgement by a certification authority.

We use the term *incremental qualification* to denote the ability to take these incremental steps in the verification effort while integrating a new component. However, there are some difficulties with a modular approach and the reuse of evidence due to concerns about safety requirements being lost. Safety is a system property and assessing the safety of components in isolation does not necessarily ensure that once integrated they will behave as expected or desired. Hence, integrated modular components need to be specified and be composed in a way in which design assurance data of a component is divided into two categories:

1) previously verified or accepted certification data which can be reused in a subsequent aircraft system design, and

2) new or revisited certification data which must be obtained in each new aircraft system design, a "delta" set of a components certification data.

In addition, details of safety analysis processes which support IMA architectures are not yet well developed. The work leading to the demonstration in this paper provides s modular, staged design and safety analysis method. It allows software components with safety interfaces to capture impact on the safety requirements in presence of faults.

Theoretical Approach

Traditional methods for safety analysis such as Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis have their deficiencies. First of all, deriving the failure propagation inside digital subsystems requires in depth knowledge about the system and becomes tedious or intractable in the absence of suitable tools. Secondly, the resulting FMEA tables and fault trees are extremely large. Thus, these methods are not optimal for incremental analysis. The effect of changing or upgrading a component in a safety-critical system often results in a complete review of the initial FMEA and FTA. This is due to the fact that the previous safety arguments must be proven to hold in the new system and there is no support for efficient incremental proofs.

One way of dealing with the increased complexity in safety assessment is to integrate the two separate activities of functional design and safety assurance through introduction of formal models that are shared and reused [2, 3]. With this approach, the safety assessment is based on the system design model and formal fault models. This model-based approach enables verification tools, such as model checkers, to automatically check if the system design tolerates the modelled failures at design time. Also, a formal framework supporting assume-guarantee reasoning can enable incremental analysis, i.e. changes in the design can be formally verified to hold by analysing parts of the system while reusing previous safety arguments. This requires that each component provides a guarantee that it behaves in a determined way in presence of faults under certain assumptions; much the same as the intuitive notion of contracts.

Earlier work has defined a formal component model supporting incremental analysis [4]. This section briefly sketches that formal approach and presents the method for incremental safety analysis. For the purpose of this paper, we will use an informal description of the formal framework; for formal definitions of components and related methodology see [4]. We will also give an overview of the tool support developed earlier.

Formal component model

The adopted underlying formalism for specifying components and component assemblies is based on the notion of reactive modules [5]. A reactive module is a model reminiscent of input output automata, for concurrent systems, that can be used for modelling both synchronous and asynchronous applications. The model supports compositional and hierarchical design which is a prerequisite for modeling complex systems. We

present a special class of reactive modules with synchronous composition and finite variable domains that we call synchronous modules (here referred as simply modules, denoted M).

A module M represents an abstract (mathematical) description of the implementation of a component. Semantically, a module can be seen as an automaton with input variables, output variables and private variables. Modules can be composed into more complex modules by the parallel operator ||, i.e. $M_1 \parallel M_2$ denotes a system consisting of the composition of the two modules M_1 and M_2. The behavior of the system consists of a set of traces, starting from the initial state and making subsequent transitions into new states based on possible transitions in the underlying automata.

Since we are focused on analysing safety of systems, we need a way to express the safety properties. A safety property φ can be seen as a set of desired behaviors i.e. traces that keep the system in desirable states. Hence, if we can prove that a module M fulfils a property φ (denoted $M \models \varphi$), we know that every behavior of the module is included in the desirable set denoted by φ. These types of proofs can for example be done automatically using model checkers.

Fault modes

In traditional safety analysis, faults can be classified into the following high-level categories: *omission faults*, *value faults*, *commission faults* and *timing faults* [4, 6, 7]. In this work, we do not focus on timing faults and our work does not include the process of *identifying* fault modes, which is itself a different research topic.

We assume a given set of fault modes (much in the spirit of FMEA), and model these faults as being part of the environment to the component, i.e. delivery of faulty input to the component, see Figure 1. Each faulty input constitutes a *fault mode* for the component. The behaviour of the fault mode is explicitly modelled as a separate module (which we denote F) that is composed in between the environment and the affected module. The input fault of one component thereby captures the output failure of a component connecting to it.

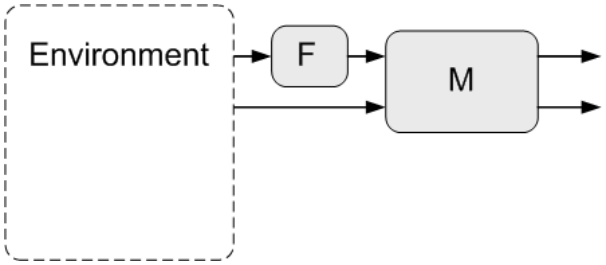

Figure 1 Fault mode composed with a model.

The composition operator for composing faults with modules is denoted ◦ and differs from the parallel operator || for technical reasons [4]. By composing the fault with the affected module we may analyse whether the safety property is fulfilled by the module even in presence of faults, i.e. whether $F_1 \circ M_1 \models \varphi$ holds. A positive result (i.e. the property holds) would imply that the module M_1 tolerates the fault F_1 (i.e. it is resilient to the fault).

Safety interface

Given a module, we wish to characterize its fault tolerance in an environment that represents the remainder of the system together with any external constraints. From a system integrator perspective, we wish to define an interface that provides all information about the component that he/she needs. Traditionally, these interfaces do not contain information about safety or fault tolerance of the component. Earlier work has defined a safety interface that captures the resilience of the component in presence of faults in the environment with respect to a given safety property φ.

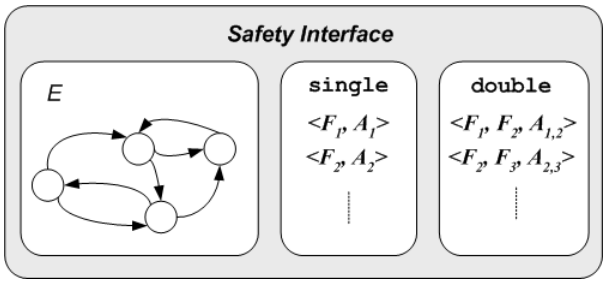

Figure 2 Informal view of a safety interface.

Figure 2 shows the components of a safety interface. Given a formal model of the behaviour of the component M,

- ***E*** gives weakest assumptions on a (fault-free) environment that ensures that component placed in this environment satisfies the safety property φ i.e:

$$E \parallel M_1 \models \varphi$$

- **`single`** describes single faults that the component is resilient to together with the assumptions (*A*'s) on the environment that need to be fulfilled in order for the component to be resilient to declared single faults
- **`double`** describes tolerated double faults together with the assumptions on the environment that need to be fulfilled in order for the component to be resilient to the listed faults

The safety interface thus makes explicit those single and double faults the component can tolerate, and the corresponding environments capture the assumptions that *M* requires for resilience to these faults. Earlier work describes how automatic generation of safety interfaces given M, fault modes, and the safety property can be supported by a front-end to formal analaysis tools [8].

A component is thus defined as a pair of elements, consisting of a module *M* describing the component's (normal) behavior, and a safety interface specifying the component's behavior in presence of faults (see Figure 3).

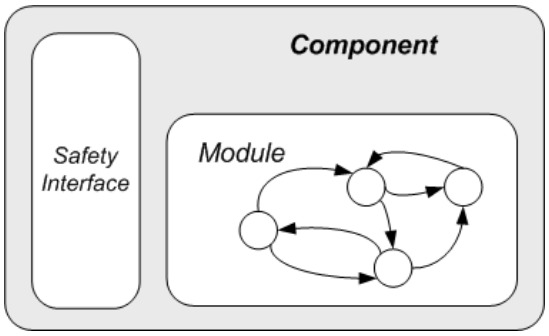

Figure 3 Informal view of a component.

Incremental Safety Analysis

Typically, safety properties are defined at system level. Thus, a safety property φ is defined on a composition of modules $M_1 \parallel M_2 \parallel ... \parallel M_n$. Hence, the straightforward way of checking that the system design satisfies the safety property (i.e. checking whether $M_1 \parallel M_2 \parallel ... \parallel M_n \models \varphi$ holds) would be to compose all components, and let a model checker analyse the system. In order to analyse the system-level fault tolerance in our setting, we could compose every module F with the affected module, i.e. $F_1 \circ M_1 \parallel M_2 \parallel ... \parallel M_n$, and check whether the safety property holds in presence of the fault using a model checker.

However, the approach of composing all modules has two main drawbacks. First of all, the composition of multiple modules may become too complex, creating a too large state space for the model checker to handle. Secondly, as described earlier, safety analysis must be done for each change in the system. Thus, if an initial analysis has been performed and a change in one component in the system is done, the system model must be (re)composed and complete analysis for all faults must be performed all over again.

This is where the safety interface can be used. As mentioned, the safety interface describes (formally) the behaviour of a component in presence of certain faults in its environment. If it may tolerate a fault, it lists the assumptions on its environment in order to tolerate the fault. By using the assumptions in an assume-guarantee reasoning framework, we are able to reason compositionally, and thereby incrementally.

Imagine that we want to check whether the system consisting of a set of modules $M_1 \parallel M_2 \parallel ... \parallel M_n$ can tolerate the single fault F_1 affecting component C_1, i.e.:

$$F_1 \circ M_1 \parallel M_2 \parallel ... \parallel M_n \models \varphi$$

The assume-guarantee rules enable us to decompose this formula into n^2 premises to check, where each individual check is less complex than the composed formula. That is, we need to show that each component pair satisfies the mutual requirements on expected environment conditions. More specifically, if F_1 is present in M_1's safety interface with assumption A_1 then we need to show that

- each of modules M_1 , M_2 ,.., M_n implement an environment that satisfies A_1.

- conversely, what each M_i ($2 \leq i \leq n$) expects from its environment is satisfied by outputs generated by $F_1 \circ M_1$

Upgrade Analysis

Reasoning is restricted to changes affected by the upgraded module and without having to redo the entire analysis each time a component changes. This results in fewer proofs on the formal design models as new components replace old ones [8].

Tool support

The above method is generic and can be incorporated in any tool chain that supports formals verification of design models and faults. Earlier work has resulted in a proof of concept tool supporting the framework as depicted in Figure 4. The generation of safety interfaces is automatically done using a front-end to Scade[9]. The output of this tool is safety interfaces for every component.

For incremental safety analysis, each individual premise is checked using the Design Verifier (in Scade).

It has also been shown that the results of the formal analysis can be presented in the form of automatically generated FMEA tables for components [10].

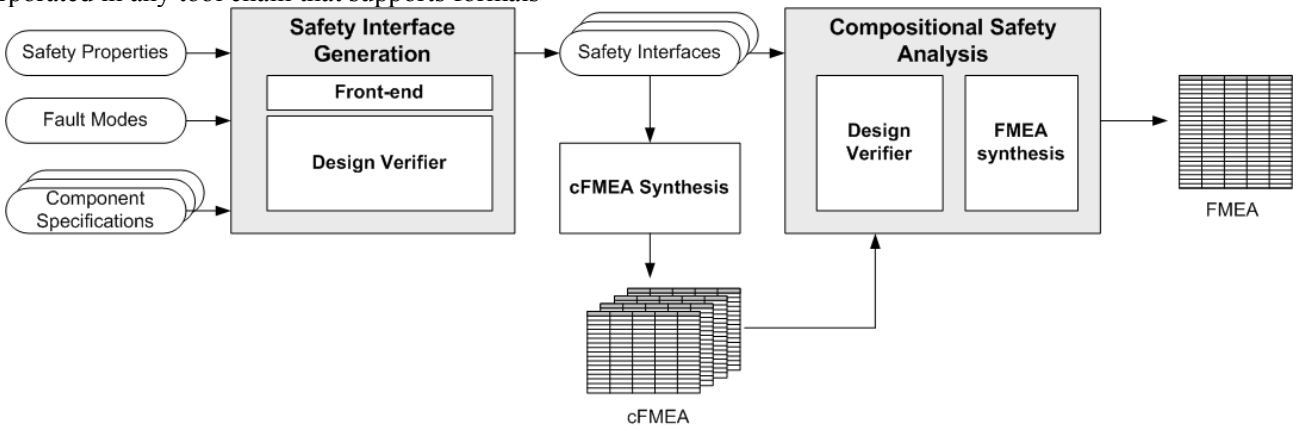

Figure 4 Tool supporting framework.

Demonstrator

The demonstrator is basically an altitude meter which autonomously controls the flap setting during take-off and landing (Auto Function and Flap System in Figure 5). The formally verified software components (Altitude and Voter in Figure 5) are integrated with the flap control function and a pressure simulator.

The goal of the demonstrator is twofold, a) the demonstrator shall be able to show that errors, handled by the functionality when performing the formal analysis, also are handled in the demonstrator after code generation. b) to demonstrate that functionality can be altered in a system, by replacing a component, without any undesired effects.

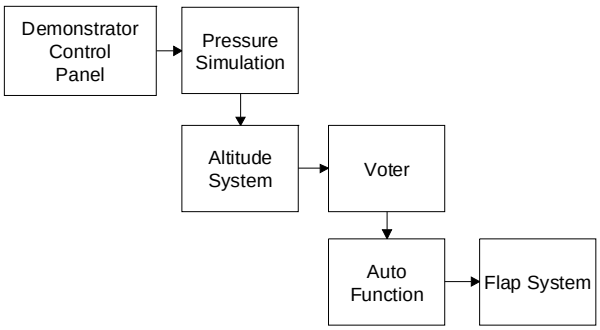

Figure 5 Functional Overview of Demonstrator.

Demonstrator Architecture

An overview of the demonstrator architecture is pictured in Figure 6, the red (large dashed) box. The test equipment consists of five parts;

- The rig control computer and rig control software
- The real-time VME system and the rig core software
- The simulation execution environment
- TTP/C communications backend
- The IMA computer under test

The rig control computer is running graphical control applications where the configurations and test cases are selected. There is also a visualization application where user created panels can be used to display various data, and control the system. The control computer is connected to the VME System via a reflective memory.

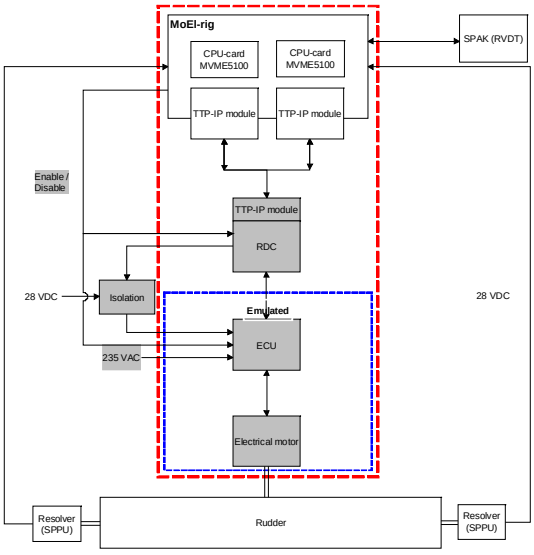

Figure 6 Demonstrator Architecture.

The VME system consists of CPU boards, running WindRiver VxWorks RTOS and handles the communication between the real-time test system and the IMA computer. It also hosts the simulation execution environment which handles the execution of simulation models, in this case the pressure simulation model. The VME system also handles the emulation of the Electric Control Unit (ECU) and the electrical motor, the blue (small dashed) box in Figure 6.

The IMA computer runs the GreenHills Integrity RTOS and handles communication to the test rig via TTP/C. The altitude system, voter, auto function and the motor control part of the flap system is executing in the IMA computer.

Demonstrator set-up

To illustrate the use of probabilistic safety interfaces we have applied the approach on an digital Altitude Meter subsystem.

The Altitude Meter subsystem calculates the altitude of an aircraft above a fixed level. Input to the Altitude Meter system is the atmospheric pressure supplied from two static ports outside the aircraft. The pressure is then transformed into a corresponding altitude. This value is then used for planning and controlling the flight. This means that the Altitude system is a safety-critical function since an incorrect value from it can have severe consequences.

The Air Data Computers (ADCs) are advanced transducers that convert the input data from the sensors (pressure) to an altitude. This is the ``pure'' altitude value, without any correction or filtering. The system consists of three ADCs, and all of them send their status as output to the IMA computer (System Computer in Figure 7).

The altitude function's goal is to filter and correct the altitude in order to get as accurate a value as possible. This is done using the air speed and the aircraft's acceleration into account. The system consists of two versions of the Altitude function, both run on the System Computer.

The role of the Voter is to compare the outputs from the Altitude Function subsystems and decide which of these values to use as output from the system.

The three ADCs (ADC1, ADC2, and ADC3) are connected directly to the System Computer. Inside, the redundancy handler checks the status of each ADC in order to detect if any of these are malfunctioning. The ADCs send a 2 bit signal, indicating ``ok'', ``degraded'' or ``total outage'', and also the calculated altitude. By checking the status of the ADCs, the Redundancy handler can choose

which of the ADC to use as primary for the Altitude Function. The Redundancy Handler also sends a status signal to the Voter.

The two Altitude Functions are both executing on the IMA computer. Input to these subsystems is the primary altitude decided by the Redundancy Handler. The altitude function filters the altitude and compensates for airspeed and vertical acceleration. Output from these are sent to the Voter.

To cope with any malfunction of the IMA computer, the altitude from the ADC3 is directly connected to the Voter with an RS-485 bus.

Safety properties and Fault modes

The safety property used here is: Altitude display shall under no conditions send incorrect altitude data (accurate to (+/-) 10m).

Following faults are included:

- Faulty pressure signal, sensor input to the ADCs (S1 / S2 in Figure 7)
- Value stuck-at fault for the data passed between ADC2 and Altitude function 2
- Backup communication channel (RS-485) error.

Table 1 Fault types

Fault	Type	Affected component
F_1	Value fault	ADC1, ADC2
F_2	Omission fault	Altitude function 2
F_3	Value fault	RS-485

Test cases

The safety interface captures the faults that the components are resilient to. The tests are carried out by injecting those faults (see Table 1).

All combinations of the above faults are tested, i.e. single, double and triple faults are tested.

The voter component exists in two different versions, one simple and one more fault-tolerant. To demonstrate the upgradability, two set-ups are used with the different Voter components.

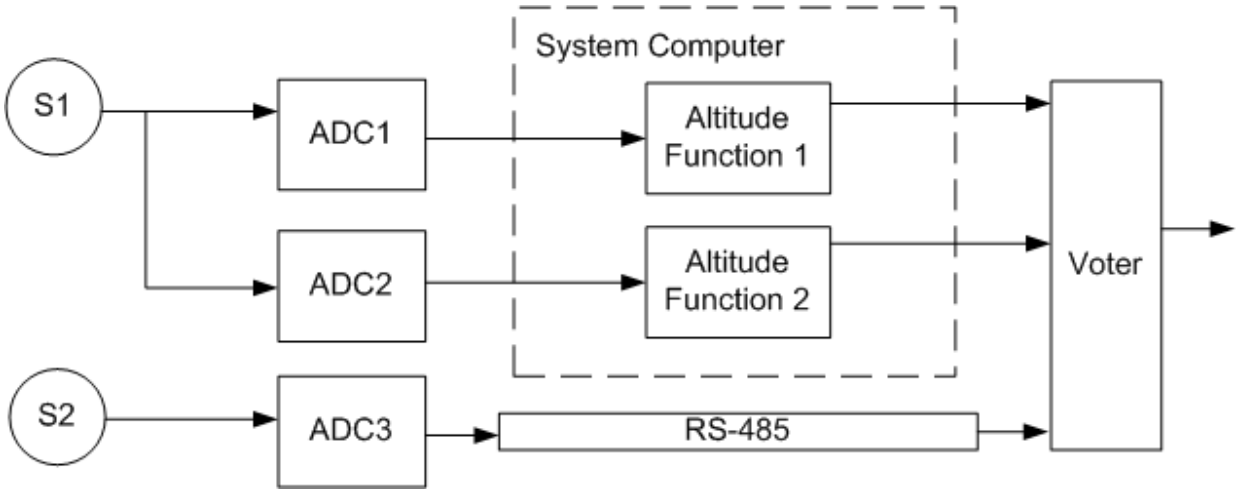

Figure 7 Altitude Subsystem and Voter.

Results and Conclusion

Initially, the safety interfaces were generated using the front-end to Scade. Using the safety interfaces, a qualitative safety analysis was performed. The result was that both F_2 and F_3 were tolerated by the system while F_1 was not tolerated (the safety property did not hold in presence of F_1). Neither were the double faults $<F_1, F_2>$, $<F_1, F_3>$ $<F2, F3>$ tolerated.

After the upgrade of the voter, the upgrade analysis showed that besides tolerating the same faults as before, also F_1 was now tolerated by the system.

These theoretical results were also demonstrated for the Altitude subsystem and Voter implemented in the test-rig. It is shown that errors, handled by the functionality when performing the formal analysis, also are handled in the demonstrator after code generation. Also it is demonstrated that functionality can be altered in a system, by replacing a component, without any undesired effects.

In addition, this work shows that it is possible to use this rather theoretical formal framework (tools, safety interface, component models, code generator etc) and integrate this into a flap control system running at an IMA computer at Saab.

Future work needs to examine how this framework regarding system safety assessment can be used.

Acknowledgements

This work was supported by the Swedish National Aerospace research program NFFP4, and project SAVE financed by the Swedish Strategic Research Foundation (SSF). The second author was partially supported by the University of Luxembourg.

References

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

[2] Hammarberg J. and S. Nadjm-Tehrani, 2005, Formal verification of fault tolerance in safety-critical reconfigurable modules, International Journal of Software Tools for Technology Transfer (STTT), vol. 7, no. 3, Springer Verlag.

[3] M. Bozzano and et al, 2003, "ESACS: an integrated methodology for design and safety analysis of complex systems," in ESREL 2003. Balkema, pp. 237–245.

[4] Elmqvist J., S. Nadjm-Tehrani and M. Minea, 2005, Safety Interfaces for Component-Based Systems, In Proceedings of 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP'05), September 2005, Springer Verlag.

[5] Rajeev Alur and Thomas A. Henzinger. 1996, Reactive modules. In Proceedings of the 11th Symposium on Logic in Computer Science (LICS '96), IEEE Computer Society, pages 207–218.

[6] Bondavalli A. and L. Simoncini, 1990, Failures Classification with Respect to Detection, In 2nd. IEEE Workshop on Future Trends in Distributed Computing Systems, 47- 53.

[7] Avizienis A., J.-C. Laprie, and B. Randell, C. Landwehr, 2004, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, Vol 1(1), pp 11-33, January 2004 .

[8] Elmqvist J. and S. Nadjm-Tehrani, 2006, Safety-Oriented Design of Component Assemblies using Safety Interfaces, Third International Workshop on Formal Aspects of Component Software (FACS'06), September 2006, Springer Verlag.

[9] Esterel Technologies. 2006, Scade Suite 4.3 User Manual.

[10] Elmqvist J. and S. Nadjm-Tehrani, 2008, Tool Support for Incremental Failure Mode and Effects Analysis of Component-Based Systems, in Design, Automation, and Test in Europe (DATE) Conference, EDA/ACM/SIGDA, München, Germany. March 2008.