



Fault-Tolerant Interactive Cockpits for Critical Applications: Overall Approach

Camille Fayollas, David Navarre, Jean-Charles Fabre, Philippe Palanque,
Yannick Deleris

► To cite this version:

Camille Fayollas, David Navarre, Jean-Charles Fabre, Philippe Palanque, Yannick Deleris. Fault-Tolerant Interactive Cockpits for Critical Applications: Overall Approach. 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012), Sep 2012, Pisa, Italy. pp.32-46, 10.1007/978-3-642-33176-3_3. hal-03647155

HAL Id: hal-03647155

<https://hal.science/hal-03647155>

Submitted on 21 Apr 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fault-Tolerant Interactive Cockpits for Critical Applications: Overall Approach

Camille Fayollas^{1,2,3}, Jean-Charles Fabre³, David Navarre²,
Philippe Palanque², and Yannick Deleris¹

¹ AIRBUS Operations, 316 Route de Bayonne, 31060, Toulouse, France
Yannick.Deleris@airbus.com

² ICS-IRIT, University of Toulouse, 118 Route de Narbonne, F-31062, Toulouse, France
{fayollas,navarre,palanque}@irit.fr

³ LAAS-CNRS, 7 avenue du colonel Roche, F-31077 Toulouse, France
Jean-Charles.Fabre@laas.fr

Abstract. The deployment of interactive facilities in avionic digital cockpits for critical applications is a challenge today. The dependability of the user interface and its related supporting software must be consistent with the criticality of the functions to be controlled. The approach proposed in this paper aims at describing how fault prevention and fault tolerance techniques can be combined to address this challenge. Following the ARINC 661 standard, a model-based development of interactive objects (namely widgets and layers) aims at providing zero-default software. Regarding remaining software faults in the underlying runtime support and also physical faults, the approach is based on fault tolerance design patterns, like self-checking components and replication techniques. The proposed solution relies on the space and time partitioning provided by the executive support following the ARINC 653 standard. Defining and designing resilient interactive cockpits is a necessity in the near future as these command and control systems provide a great opportunity to improve maintenance and evolutivity of avionic systems.

Keywords: Interactive Systems, Self Checking Components, Widgets, Dependability, Fault Tolerance, Resilient Computing.

1 Introduction

In the late 70's the aviation industry developed a new kind of display system integrating multiple displays and known as "glass cockpit". Using integrated displays it was possible to gather under a single screen a lot of information usually statically attached to multiple independent traditional mechanical gauges. This generation of glass cockpits uses several displays with electronic technology. It receives information from aircraft system applications of embedded equipment, then processes and displays this information to crew members. In order to send controls to the aircraft systems, the crew members have to use physical buttons usually made available next to the displays. Control and display are processed independently (different hardware and

software) and without integration. This is something that is about to change radically with interactive cockpit applications which have started to replace (e.g. with Airbus A380) the glass cockpit. The main reason is that it makes possible to integrate information from several aircraft systems in a single user interface in the cockpit. This integration is nowadays required in order to allow flight crew to handle the always increasing number of instruments which manage more and more complex information. Such integration takes place through interactive applications featuring graphical input and output devices and interaction techniques as in any other interactive contexts (web applications, games, home entertainment, mobile devices ...). The interactive system in the cockpit is called Control and Display System (CDS). It performs various operational services of major importance for flight crew and allows the display of aircraft parameters using LCD screens (as in the glass cockpit), but it also allows the pilots to graphically interact with these parameters using a keyboard and a mouse to control aircraft systems.

If the information displayed on (or controlled by) the CDS is critical, its general architecture must be fault-tolerant. Similarly, on the user side, the information flow between the pilot and the CDS has to ensure safe flight operations, avoiding the entering of wrong or incomplete data and displaying only correct information (with respect to the inner state of the aircraft). Ensuring such dependability of information is not an easy task due to the high number of hardware and software components involved.

In this paper, we propose an approach for assessing dependability of interactive cockpits. The approach is based on conventional fault tolerance techniques applied to interactive objects, at various abstraction levels. The approach also relies on the error confinement facilities provided by the runtime support, namely an ARINC 653 operating system kernel [2]. Such runtime platform provides both space and time partitioning. Today the interaction is devoted to non-critical avionic functions and the management of the interaction is located into one partition for the whole interactive system. The basic idea is to design fault tolerant interactive entities using multiple partitions providing error confinement on redundant Display Units.

The paper is organized as follows. In section 2 we describe the problem statement in more details before illustrating the architecture of interactive cockpits using the A380 example in section 3. We put focus in this section to the organization of a digital interactive system following the ARINC 661 standard [1]. In section 4, we describe our approach to improve the dependability of interactive systems for avionic critical applications. The architecture of our fault tolerant solution is described in section 5 and is based on a conventional approach for fault tolerant computing in avionics, the COM-MON approach [15]. Section 6 concludes this paper.

2 Problem Statement

It is worth noting that, currently, the interactive approach is only used in avionics for non-safety critical functions. The challenge is now to use this interactive approach for critical functions.

The interactive system can be viewed at various levels of abstraction, from individual widgets at the bottom level up to more complex interactive entities associated to so-called user applications, namely the interactive counterpart of avionic functions. The specifications of interactive objects (widgets and layers) are defined in the ARINC 661 standard. Such complex system is subject to faults that can impair the correct rendering of information and the delivery of input information to avionic functions. In this work, we focus on dependability issues of the CDS, as a computer-based system so the type of faults considered ranges from software to hardware faults and we do not consider human errors. We understand that such errors are definitely of interest as far as man-machine interface and interactive systems are considered. However the evolution of conventional cockpits to total digital cockpits is a revolution and raises the challenge of its dependability with respect to conventional software and hardware faults. The dependability of such complex computer-based systems should be as high as possible depending on the associated failure cases considered: as far as critical functions are concerned, the system must be developed in compliance with the highest assurance level, so-called DAL A (Design Assurance Level A) according to the D0178B development process standard [15].

The approach proposed in this work is two-fold and relies both on fault prevention and fault tolerance approaches. To deal with software faults, we advocate the use of a model-based approach of interactive applications, limiting as far as possible software faults in the resulting interactive entities. A formalism based on Petri Nets is used to formalize the specifications of interactive objects (widgets and layers). This formal representation of the interactive objects is interpreted at runtime by a virtual machine. Regarding physical faults, we advocated a fault tolerance approach relying on the *N-Self Checking Programming* paradigm [3] and taking advantage of the ARINC 653 features and redundant hardware.

The long-term interest of such digital cockpit is its flexibility. The lifetime of a civil aircraft is about 40 years and is subject to many evolutions and updates. This digital cockpit approach vs. the physical approach offers a clear benefit in this respect, provided the dependability of the interactive system is guaranteed when changes occur. This calls thus for resilient computing [15] solutions by definition, which is the long-term aim of this work.

3 Interactive Cockpit Architecture

3.1 Cockpit Architecture Overview: An Example

In this paper, we will take the example of the Airbus A380 cockpit (see **Fig. 1**). The interactive control and display system (CDS) of the Airbus A380 is composed of 2 input devices called KCCUs (Keyboard and Cursor Control Unit) and 8 output devices called DUs (Display Unit). Only some of the 8 DUs allow the crew to use the interactivity, the other ones are only used for displaying information.

A DU device is composed of a LCD screen, a graphic processing unit and a central processing unit running an ARINC 653 [2] operating system kernel. The software responsible for the interactivity is processed in the DU within one partition.



Fig. 1. Airbus A380 interactive cockpit

The type of user interfaces that can be proposed on these DUs is based on ARINC 661 specification [1]. Beyond that user interface aspect, ARINC 661 specification also defines the communication protocol (see **Fig. 2(c)**) between the various architectural components of an interactive cockpit (see **Fig. 2**). The ARINC 661 is based on client-server architecture.

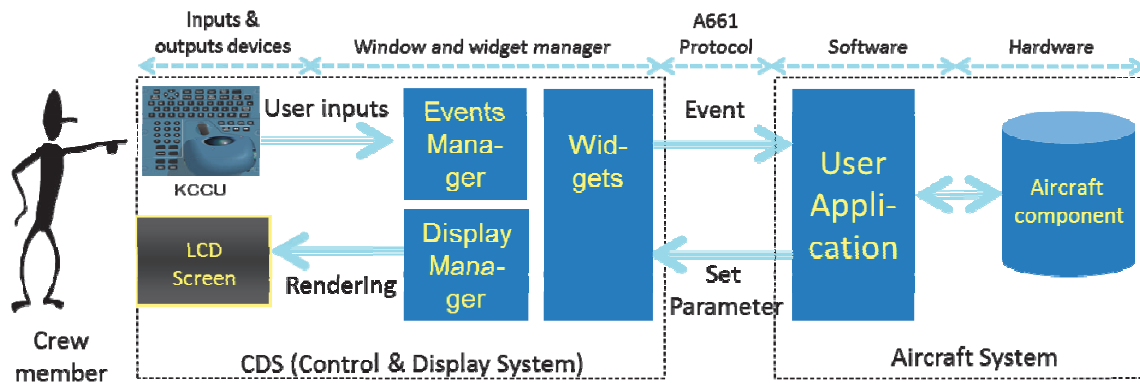


Fig. 2. Simplified Architecture compliant with ARINC 661 specification standard : (a) Crew; (b) CDS, (c) ARINC 661 Protocol; (d) Aircraft System

The server is part of the CDS (Control & Display System) in **Fig. 2** (b) and it is composed of the following elements:

- *Input and output devices*: KCCUs (Keyboard and Cursor Control Unit) and LCD screens. They allow crew members to interact with the application.
- *Window and widget managers*: composed of an event manager, a display manager and a set of graphical elements (called widgets) distributed in a set of windows rendered on the LCD screens. For instance, the CDS is responsible for handling the creation of widgets, managing KCCU graphical cursors, dispatching the events to the corresponding widgets and the rendering of graphical information on the LCD screens.

The CDS manages information for two types of clients:

- *Aircraft Systems* (**Fig. 2(d)**): information to and from aircraft systems flows through dedicated so-called User Applications (UAs) which are applications featuring a graphical user interface for a given avionic function. They process the event notifications sent by the widgets (and might trigger commands on the physical aircraft components). They can also update the widgets (by calling update methods called `SetParameters`) in order to provide feedback to the flight crew according to state changes which occurred in the aircraft systems.
- *Crew members* (**Fig. 2 (a)**): they have the responsibility of flying the aircraft by monitoring the aircraft systems through the LCD screens and controlling the aircraft system through input devices. They interact with the displayed widgets. For instance, they can click on a button in order to trigger a command, enter a numeric value in an `EditBoxNumeric` to send a value to an avionic function.

3.2 Interactive Software Organisation

The cockpit interactive user interfaces use a windowing concept (see **Fig. 3**) that can be compared to a desktop computer system windowing.

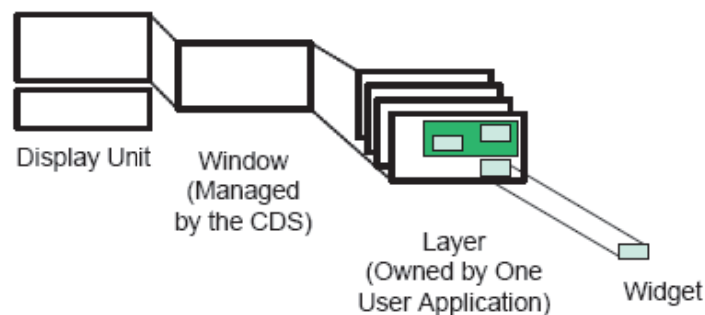





Fig. 3. ARINC 661 windowing concept

Each DU display surface is divided into windows. Each window can be subdivided in one or more layers. A layer is associated to one UA and represents the display and the interactive facilities required by an avionic function. The UA and the CDS share data within this layer and its hierarchical structure. The layer hierarchical structure can be seen as a widget tree. The layer is the highest level of this tree. The widgets are the basic interactive components such as:

- Pushbutton: dedicated to commands triggering. 
- Radio buttons: selection of one option amongst a set of available ones. 
- `EditBoxNumeric`: for entering numeric values. 

The construction of the structural widget tree is made possible by special widgets called *containers* that can contain other widgets. We called the containers *parents* and the widgets they contain *children*.

4 Overall Approach

In this section, we first present the main assumptions we make, the functional failures we want to prevent and the fault model we consider to select the appropriate fault tolerance strategies. Then we propose our approach for embedding dependability mechanisms within an interactive cockpit. Our approach is two-fold, we first use a model-based approach to develop our software and deal with software faults, then we introduce well-known dependability mechanisms to deal with physical faults.

4.1 Main Hypotheses and Functional Failures to Cover

The focus of this paper is on the interactive system dependability, more precisely, the CDS dependability. Human-errors are out of the scope of our study, the target being here the dependability of CDS as a computer-based system. As the CDS is a really large and complex entity, we decided to focus first on the server reliability. To concentrate on this problem, we assume the following:

- The communication between the CDS and aircraft system is reliable. The data transfer is without corruption and this can easily be achieved using conventional reliable protocols on a FIFO communication channel.
- The reliability of user applications (UA), the display related part of an avionic function, is out of the scope of this work, we consider that all information received by the CDS from aircraft systems is correct.
- The displays of the CDS are reliable, graphical commands sent to the LCD screen are always correctly displayed.
- The KCCU is sending reliable data to the server.

Our main interest is to ensure that the server processes correctly input events from crew members, and send graphical commands to the LCD screen according to the data received from user applications. Three possible failures must be avoided:

- **Erroneous Display:** Transmission of an erroneous value to the display according to the data received from aircraft systems (e.g. a widget receives the value x to render and transmits to the display another value);
- **Erroneous Control:** Transmission of a different action from the one done by crew members (e.g. a crew member clicks on Button1 but the event Click-Button2 is sent to the application);
- **Inadvertent Control:** Transmission of an action without any crew members' action (e.g. an event click is sent to the application without crew action on input devices).

The fault model considered in our study encompasses physical faults ranging from crash faults, due to a power supply failure of an electronic board for instance, to more subtle faults like Single Event Effects [15]. Regarding software faults, the model-based design approach proposed in the next section aims at limiting very much the introduction of design faults in the development process. Furthermore, to consider transient

software faults the interactive software and the base executive software, namely the ARINC 653 kernel will be developed at the highest assurance level (DAL A) and thus considered as a zero-default piece of software.

4.2 Using ICO Formal Modeling to Design Interactive Cockpits

In the domain of the design of safety-critical interactive systems, the use of a formal specification technique is extremely valuable because it provides non-ambiguous, complete and concise models. The advantages of using such formalisms are widened if they are provided with formal analysis techniques that allow proving properties about the design [3], thus giving an early verification to the designer before the application is actually implemented [5].

The Interactive Cooperative Objects (ICO) is a formal description technique dedicated to the specification and verification of interactive systems [11]. It uses concepts borrowed from the object-oriented approach (dynamic instantiation, classification, encapsulation, inheritance, client/server relationship) to describe the structural or static aspects of interactive systems, and uses high-level Petri nets [7] to describe their dynamic or behavioural aspects. As an extension of the Cooperative Objects formalisms it has been designed to describe behavioural aspects of objects-based distributed systems [4]. The formalism is able to handle the specific aspects of interactive systems. In a nutshell, the ICO formalism can be described as follows:

- ICO is Petri net based, suitable to specify the behaviour of event driven-interactive systems and concurrent human-computer interactions, but also able to describe the inner states of the Interactive Application.
- The formalism enables the handling of more complex data structure (typed places and tokens, transitions with actions and preconditions, variable names on arcs).
- ICO objects react to external events according to their internal state and they can produce events.
- An object is defined as the set of four elements: an extended Petri Net describing the behaviour of the interactive object, a presentation part, and two functions (the activation function and the rendering function) that make the link between the cooperative object and the presentation part (events from input devices and output on the LCD screens).

In previous work [3], we have proposed the use of ICO formal modeling for describing in a complete and unambiguous way both standard widgets and interactive applications following ARINC 661 specifications.

Any widget corresponds to a collection of interconnected Petri Nets. For instance, the ICO model of the `PicturePushButton` (PPB) may be divided in 7 sub-parts (one handling mouse click events and the other 6 for managing one parameter each: *Visible*, *Enable*, *PictureReference*, *LabelString*, *StyleSet*, *Highlighted*).

To illustrate the model, we show in **Fig. 4** the handling of mouse click events. The `PicturePushButton` has two internal states: it can be (i) pressed or (ii) released. State changes are due to user actions (mouse down events, mouse click events). The mouse click events are relayed to the widget via the `processMouseClicked` method. They are handled only if

the widget is enable and visible. In this case, the event A661_EvtSelection is triggered; otherwise, the mouse click events are discarded. **Fig. 4** is only a small part of the entire PicturePushButton ICO model, which is composed of 37 places and 24 transitions. It is modeled as (i) the various states it can be in (e.g. visible, enable, pressed), (ii) the set of method calls he can process (e.g. processMouseClicked, setLabelString), (iii) the set of events it can trigger (e.g. A661_EvtSelection) and (iv) when such events are triggered (see **Fig. 4**). The entire behavioral description of the PPB can be found in [14].

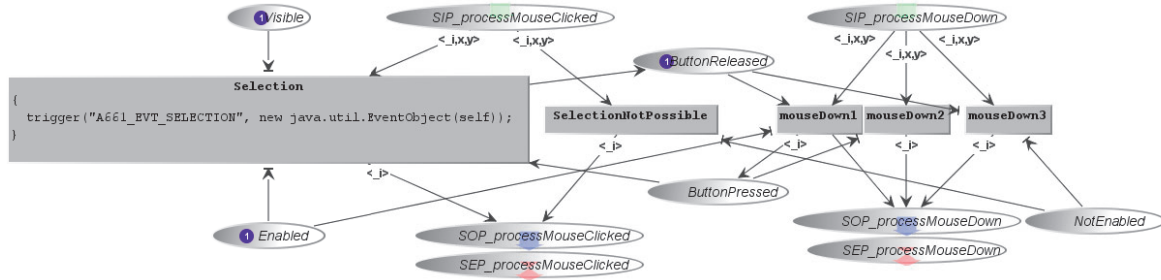


Fig. 4. ICO model of the management of mouse click events for the PicturePushButton

4.3 Introducing Dependability Mechanisms into Interactive Cockpits

Many dependability strategies rely on the notion of self-checking component as introduced in [3] and [15], ideally grouping a function and its corresponding controller. A self-checking component provides error-confinement and can be thus considered as a fail-stop component. The error detection coverage of a self-checking component is expected to be very high, making thus replication strategies possible to provide service continuity. This concept has been used for many safety-critical functions in avionic systems: the COM/MON approach [15] is the basis for various N-Self-Checking Programming (NSCP)-based architectures [3]. We decided to apply the self-checking approach to two abstraction levels of our interactive system, the first one being the widgets, the second one being the layers.

According to [3], "a self-checking software component consists of either a variant and an acceptance test or two variants and a comparison algorithm". In accordance with this definition, we can generalize a self-checking component to a *functional component* associated with a *controller* or *checker*. Then, several options can be considered for the implementation of a self-checking interactive object:

- **Option 1:** A copy of the functional component (called *controller* or *checker*) and a voting mechanism (called *comparator*).
- **Option 2:** A diversified variant of the functional component (as a *controller*) and a voting mechanism (the *comparator*)
- **Option 3:** A safety properties checker, the controller being responsible in this case for the verification of a number of safety properties associated to the interactive object semantics in its operational context.

In the two first self-checking options, the functional and the controller components are processing inputs at the same time, the comparator then compares both outputs

and sends an error if the functional outputs and the controller ones are different. Both options tolerate transient software faults, the second aiming at tolerating design faults.

In the last self-checking options, the safety properties checker checks some properties defined as safety ones. We check if the outputs are consistent with the inputs. This last option tolerates transient faults or remaining design faults that impair the safety properties.

Self-Checking Widgets

As a start, we have used the option 1 to implement self-checking widgets. A self-checking widget [14] is made up of 5 interconnected components (see **Fig. 5**):

- **The self-checking widget** (or façade) is the global widget, coordinating the data flow to and from the other sub-components. This encapsulation of the other inner components makes it possible to hide (as much as possible) the self-checking nature of the component which can interact with the rest of the application.
- **The dispatcher:** events received by the self-checking widget are received by the dispatcher. The dispatcher duplicates this event and sends it both to the functional and controller using a simple atomic broadcast protocol (all or nothing semantics).
- **The functional** component is the behavioral model of the non-fault-tolerant widget. The outputs are sent both to the self-checking widget and the comparator.
- **The controller** is a second version of the widget. It only implements the functionalities that have to be supervised by the controller. The controller sends its output to the comparator.
- **The comparator** is in charge of comparing the functional and controller outputs.

The dispatcher and the comparator have obviously important roles and should be zero-default. They are quite simple and are subject to intensive testing.

The comparator raises errors that may invalidate outputs as shown in **Fig. 5**. Two kinds of comparison that can be performed: one related to parameters modification and the other related to event notification. When the comparator receives an output from the functional component (resp. the controller) it waits for the corresponding output from the controller (resp. the functional component). Following the reception of these two outputs, 3 types of errors can occur: (i) one of the outputs is not received, (ii) one of the outputs is received too late with respect to the defined temporal window, (iii) the outputs don't carry the same value. In case of error, the comparator raises an error event.

One of the key aspects of the proposed architecture is that it allows the segregation of the five sub-components (e.g. each sub-component may be executed on different processors with different resources). Indeed, a self-checking mechanism is not enough to ensure fault-tolerance if a fault occurring on one component might interfere with the behavior of another component. This would be the case if all the components of the architecture were executed in the same partition. ARINC 653 [2] defines such partitioning in avionic systems and our contribution relies on this notion (section 5).

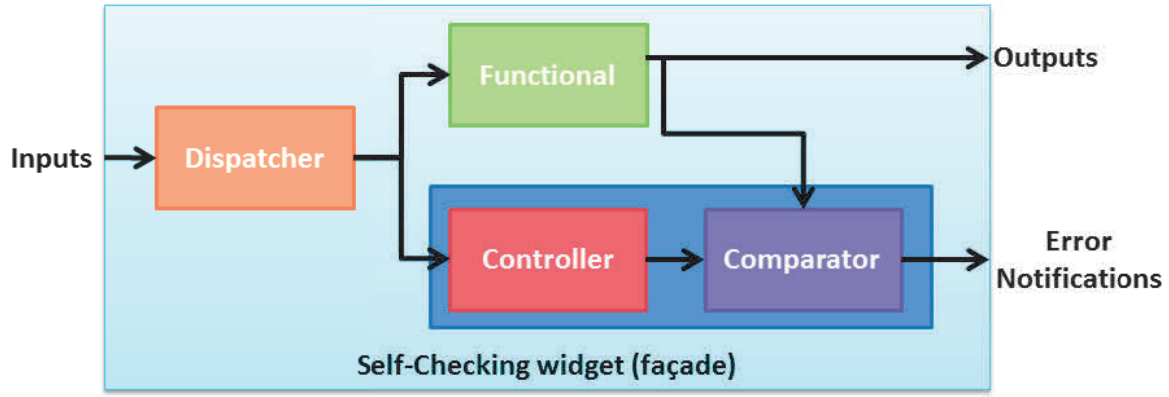


Fig. 5. Self-checking widget functional architecture

As presented in **Fig. 5**, adding fault-tolerance mechanism to the `PicturePushButton` is the result of the merge of the five subparts of the self-checking component architecture (the self-checking-component façade, the dispatcher, the functional, i.e. the classic non-self-checking widget as presented in section 4.3, the controller and the comparator). Each component is modeled in a different Petri Net model. For lack of space, we will not present these components here.

Self-Checking Layers

The approach presented above is very generic as it can be applied to each widget, without any knowledge of the application: as explained in section 3, the widget is the basic interactive component. Therefore, this approach can be resource consuming. Indeed, to cover both transient and permanent hardware faults, we need to isolate the functional component and its controller in different partitions (see section 5). In addition, self-checking interactive objects must be replicated on different display units, following the principle of N-Self Checking Programming. Four partitions on 2 DUs for every widget does not seem acceptable due to resource overheads. It is worth noting that an interactive application can contain a lot of widgets, the partition number will then be really very high (a standard user application requires about several hundreds of widgets) even if not every widgets will be considered as critical and then will need to be fault-tolerant.

To solve this issue, we propose to apply the self-checking mechanism to an upper level: the layer. The layer is a logical unit merging all the widgets for one application, for one UA. In this case, the objective is not to validate every output at the widget level, but safety properties related to a sequence of interactions between the layer and its attached UA. The granularity of the verification is done in this case at a more abstract and semantic level.

To illustrate more concretely a layer, we give an example in **Fig. 6** containing one layer grouping 138 widgets: it is an interactive cockpit application called Flight Control Unit (FCU) Backup. The FCU is a hardware panel (i.e. several electronic devices such as buttons, knobs, displays ...) providing two types of services: Electronic Flight Information System (EFIS) and Auto Flight System (AFS).

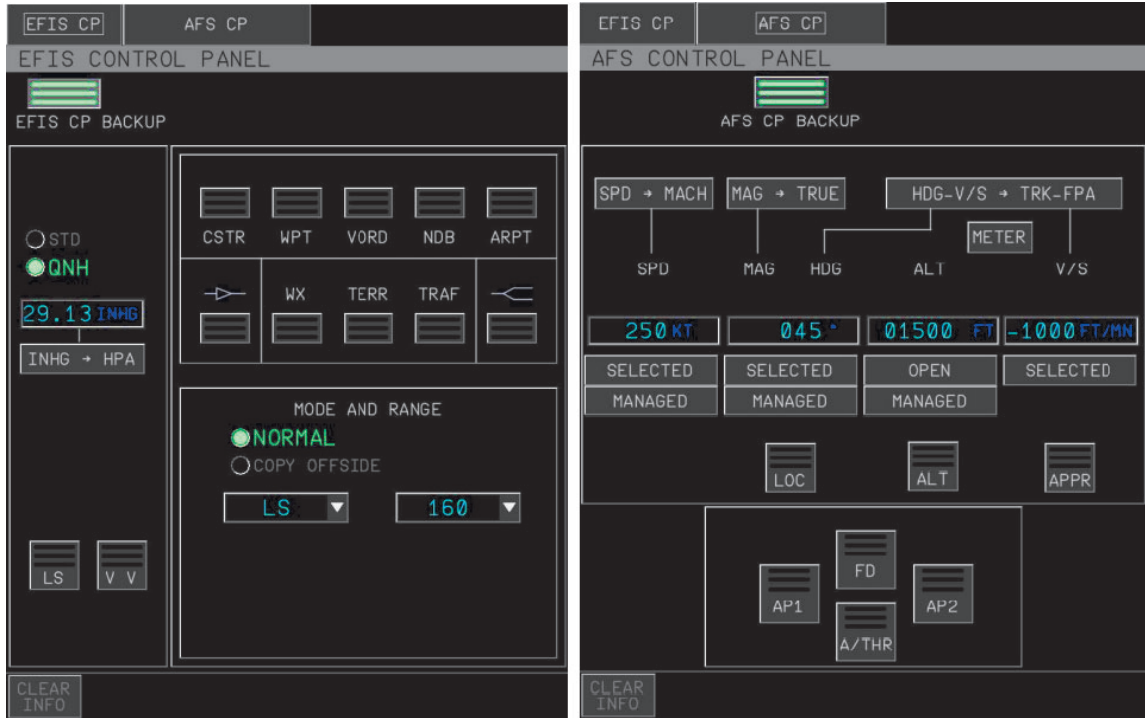


Fig. 6. Snapshot of the FCU Backup application in Airbus A380 (left EFIS, right AFS)

The FCU Backup application (see **Fig. 6**) is designed as an ARINC 661 layer to recover all FCU functions in case of failure of the physical FCU. It is composed of two interactive pages: EFIS Control Panel (CP) and AFS CP and is displayed on two of the eight LCD screens (for the A380) in the cockpit, one for the Captain and the other for the First Officer. Both crew members can interact with the application via the KCCUs which gathers in the same hardware component a keyboard and a trackball.

As the layer is directly connected to an application, it is easy to define safety assertions associated with the application semantics. Beyond self-checking widgets and generic sequences of widget actions, we have to consider safety properties that can be checked.

A simple example can be found on the FCU-Backup application (see **Fig. 6**). The `EditBoxNumeric` on the left of EFIS CP displays the atmospheric pressure. It can be displayed in two different units (inHg or hPa). The atmospheric pressure unit is chosen using the `PicturePushButton` just behind the `EditBoxNumeric`. A safety property correspond to the verification that the unit and value are modified in a right way upon a click on the `PicturePushButton`.

A self-checking layer implementation relies in part on a controller as a *safety properties checker* (option 3 defined in section 4.3). Furthermore, as explained previously, the layer is a very big and complex entity and thus very complicated to model. The application of the self-checking pattern to a layer is of interest when safety properties have to be checked, because of their semantic nature.

5 Fault-Tolerant Architecture

The self-checking interactive objects (widgets and layers) aim at improving the error detection coverage regarding the fault model described previously. The execution of

an interactive object developed using ICO relies on several software layers: a Petri Net simulator (e.g. PetShop [11] in our case) or code generated from the model, a virtual machine (a JVM in our case), display and event managers belonging to the CDS and at last the ARINC 653 operating system kernel.

To cover both transient faults and permanent hardware faults, it is mandatory to take into account error confinement areas to isolate the functional part and the controller part of the self-checking object, whatever it is a widget or a layer.

5.1 Architectural Issues

Ideally the functional part should be located in one partition, the controller in a second partition and the dispatcher and the comparator in a third partition. A simplification can be to locate the controller, the dispatcher and the comparator in a single partition, the three components being considered as a verification logic. A partition providing space and time segregation prevent faults having an impact on both the function and its controller counterpart.

To tolerate crash faults, two copies of the self-checking widget should be located on two different DUs, as a physical unit. Only one is considered active at a given point in time. This approach follows the *N-Self-Checking Components* principle early mentioned in this paper. Because interactive objects hold a persistent state, a *master-slave* replication strategy is mandatory. Two design patterns of duplex protocols can be envisaged: a checkpointing-based strategy (primary-backup replication protocol) or an active replication strategy (e.g. a leader-follower replication protocol). In short the architecture can be sketched as follows:

- a) the functional part F1 of the interactive object is located in P1 on DU1
- b) the controller part C1 of the interactive object is located in P1 on DU2
- c) a replica of the functional part F2 is located on P2 on the DU2
- d) the controller part C2 of the functional part replica is located in P2 on DU1

Fig. 7 illustrates these implementation choices: F1/C1 is the master and F2/C2 is the slave. The slave does not interact with the crew, only the master does. This means that inputs from the crew on the master ICO object are forwarded to the slave object. Events produced by the slave, if it is an active copy, are not forwarded to the UA, only events from the master are delivered to the UA. More details on design patterns for resilient computing can be found in [8].

Each partition contains an ARINC 661 server, implemented as an ICO model allowing the communication between the ICO interactive object (layer or widget) and the UA or the input and output devices. In order to execute ICO models all partitions include a JVM on top of which our Petshop tool is running. This is the option we consider now in our experiments, which is conformant to the fault assumptions early described. This option does not consider remaining development faults within the Petshop tool or the JVM, i.e. common mode faults in the executive software. We come back to this point in the next section.

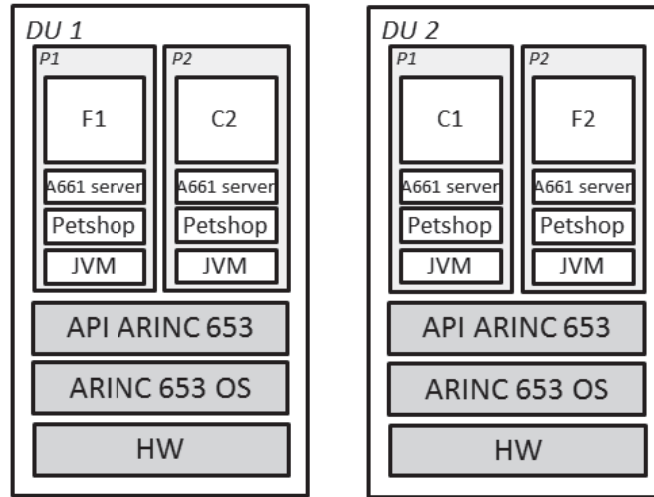


Fig. 7. Fault tolerant architecture

It is worth noting that the proposed fault tolerant architecture implementation is using 2 DUs. Yet, depending on the requirements, other implementations with 3 or more DUs can also be studied based on this principle.

5.2 Performance Issues: Discussion

The performance of the architectural solution proposed in the previous section can be analyzed through different angles: dependability and resource overheads.

From a dependability viewpoint, the proposed solution relies first on the use of a model-based development that minimizes the introduction of design faults. The ARINC 661 standard defines the behavior of the widgets and a non-ambiguous description is obtained using the ICO formalism. The model is interpreted at runtime and implements the expected interactive objects for a given avionic function through a so-called user application (UA). The communication between the interactive object and the UA or the input and output devices is insured by an ARINC 661 server modeled in ICO. The execution relies on a specific Petri Net interpreter running on top of a JVM. The remaining design faults in the executive support (Petshop, the JVM, or even the server) can be a problem for critical applications. To address this issue, a first solution can be to use the Petri Net interpreter in one partition for the functional part (F1/P1/DU1 in **Fig. 7**) and generate code for the controller (C1/P1/DU2). The controller is a copy of the functional part, but in this case the runtime image is different. One can also consider using several implementations of the JVM (diversification) in different partitions. A similar approach based on different ADA runtime supports has been used in the B777 architecture [17].

Whatever the option is, the space and time partitioning provided by the ARINC 653 kernel clearly isolates the functional part of a self-checking interactive object from other ICO objects running in different partitions. Errors due to memory faults and infinite loops have no side effects on ICO objects running in companion partitions on the same DU. In options 2 and 3 proposed in Section 4.3, the controller differs from the functional interactive objects. A reduced version of the functional interactive

object specification can be implemented as a controller if some aspects of the ICO object have no impact on dependability and can be ignored. Moving forward with this approach leads in fact to the third option, where the controller only checks safety properties, i.e. executes user-defined executable assertions. A white box approach enables assertions to benefit from deeper observability of the ICO object behavior.

Whatever the implementation option is (widget or layer), the resource overhead (timing, communication, etc.) has to be considered. We plan to provide measures related to (i) the complexity of the model (number of states and transitions) but also (ii) to communication overheads (number and size of messages) between functional and controller partitions.

6 Conclusion and Perspectives

In this paper, we have shown that safety critical applications (such as interactive cockpits applications) raise specific concerns with regard to fault-tolerance and resilience. We have presented an approach to increase safety critical interactive system resilience by enriching them with fault-tolerance mechanisms. We proposed to introduce a self-checking mechanism at two abstraction levels of the interactive system: the widget and the layer.

These two approaches can be used separately or jointly. The design choice (self-checking widget or self-checking layer) is left open to the UA designer according to the criticality of the avionic function considered in general, but also with respect to the criticality of the parameter (or event) to be obtained or the parameter to be displayed. For instance, the UA designer can choose to use the layer approach yet, he can use jointly the widget approach for really critical information. We also presented an architecture compliant with our approach. To go further, we are currently applying our approach to the FCU Backup application mentioned in the paper.

The approach presented in this paper has been implemented in Java as a first proof of concepts, but the final implementation might be different. We are currently investigating in more details the notion of self-checking layer and plan to implement the FCU Backup case study on a realistic platform.

Acknowledgment. This work is partly funded by Airbus under the contract R&T Display System X31WD1107313.

References

1. ARINC 661 Cockpit Display System Interfaces to User Systems. ARINC Specification 661. Airlines Electronic Engineering Committee (2002)
2. ARINC 653 Avionics Application Software Standard Interface. ARINC Specification 653. Airlines Electronic Engineering Committee, July 15 (2003)
3. Barboni, E., Conversy, S., Navarre, D., Palanque, P.: Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In: Doherty, G., Blandford, A. (eds.) DSVIS 2006. LNCS, vol. 4323, pp. 25–38. Springer, Heidelberg (2007)

4. Bastide, R., Sy, O., Palanque, P.: A formal notation and tool for the engineering of CORBA systems. *Concurrency: Practice and Experience* (Wiley) 12, 1379–1403 (2000)
5. Degani, A., Heymann, M.: Analysis and Verification of Human-Automation Interfaces. *Human Centered Computing: Cognitive, Social and Ergonomic Aspects*. In: *Proceedings of the 10th Int. Conf. on HCI*, vol. 3, pp. 185–189. Erlbaum, Mahwah (2003)
6. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc., EUROCAE (December 1992)
7. Genrich, H.J.: Predicate/Transitions Nets. In: Jensen, K., Rozenberg, G. (eds.) *High-Levels Petri Nets: Theory and Application*, pp. 3–43. Springer, Heidelberg (1991)
8. Gibert, V., Machin, M., Fabre, J.-C., Stoicescu, M.: Design for Adaptation of Fault Tolerance Strategies. *Rapport LAAS no 12198*, 35 p (April 2012)
9. Laprie, J.-C.: From Dependability to Resilience. In: *IEEE/IFIP International Conference on Dependable Systems and Networks*, Anchorage, Alaska, USA (June 2008)
10. Laprie, J.-C., Arlat, J., Béounes, C., Kanoun, K.: Definition and Analysis of hardware and software Fault-Tolerant Architectures. *IEEE Computer* 23(7), 39–51 (1990)
11. Navarre, D., Palanque, P., Bastide, R.: A Tool-Supported Design Framework for Safety Critical Interactive Systems in Interacting with computers, vol. 15/3, pp. 309–328. Elsevier, Amsterdam (2003)
12. Navarre, D., Palanque, P., Ladry, J.-F., Barboni, E.: ICOs: a Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. *ACM Trans. on Computer-Human Interaction* 16(4), 1–56 (2009)
13. Normand, E.: Single-event effects in avionics. *IEEE Transactions on Nuclear Science* 43(2), 461–474 (1996)
14. Tankeu-Choitat, A., Navarre, D., Palanque, P., Deleris, Y., Fabre, J.-C., Fayollas, C.: Self-checking components for dependable interactive cockpits using formal description techniques. In: *Proc. of 17th IEEE Pacific Rim Int. Symp. on Dependable Computing (PRDC 2011)*, Pasadena, California, USA (2011)
15. Traverse, P., Lacaze, I., Souyris, J.: Airbus Fly-by-Wire: A Total Approach to Dependability. In: *Proceedings 18th IFIP World Computer Congress, Building the Information Society*, Toulouse, France, August 22–27, pp. 191–212 (2004)
16. Yau, S.S., Cheung, R.C.: Design of self-Checking Software. In: *Proc. Int. Conf. on Reliable Software*, pp. 450–457. IEEE Computer Society Press, Los Angeles (1975)
17. Yeh, Y.C. (Bob): Design Considerations in Boeing 777 Fly-By-Wire Computers. In: *Third IEEE International High-Assurance Systems Engineering Symposium*, p. 64 (1998)