A Fault-Tolerant Software Architecture and its Formal Specification for Embedded, Real-Time Interactive Systems

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

A Fault-Tolerant Software Architecture and its Formal Specification for Embedded, Real-Time Interactive Systems

C. Fayollas[2,3], P. Palanque [2], J.-C. Fabre[3,4], D. Navarre[2], Y. Deleris[1], A. Hamon[1,2]
[1] AIRBUS Operations, 316 Route de Bayonne, 31060, Toulouse, France
[2] ICS-IRIT, University of Toulouse, 118 Route de Narbonne, F-31062, Toulouse, France
[3] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[4] Université de Toulouse, INP, LAAS, F-31400 Toulouse, France
(fayollas, navarre, palanque, hamon)@ irit.fr, yannick.deleris@airbus.com, Jean-Charles.Fabre@laas.fr

***Abstract*—Most of the work that has been done to build reliable interactive systems has been focusing on avoiding the occurrence of faults during the development of the system, using for instance formal verification techniques. However, empirical studies have demonstrated that software crashes may occur at runtime, even if the development has been extremely rigorous. One of the many sources of such crashes is called natural faults triggered by alpha-particles from radioactive contaminants in the chips or neutron from cosmic radiation. A higher probability of occurrence of faults concerns systems deployed in the high atmosphere (e.g. aircrafts) or in space (e.g. manned spacecraft). Therefore mechanisms are needed to deal with these faults and guarantee that the system will work correctly even in the presence of these faults. To deal with this issue, this paper proposes a fault-tolerant software architecture and its formal specification applied to embedded, real-time interactive systems.**

Keywords—Dependability, Widgets, Fault Tolerance, Formal Description Techniques, Interactive cockpits, Fault-Tolerant Architecture

I. INTRODUCTION

A safety-critical system is a system in which any failure or error has the potential to lead to loss of lives or to injury human beings [13] while a system is called critical when the cost of a potential error is much higher than the cost of development. Whether or not they are classified as safety-critical or critical, interactive systems have made their way into most of the command and control workstations including satellite ground segments, military and civil cockpits, air traffic control... The complexity and quantity of data manipulated, the amount of systems to be controlled and the high number of commands to be triggered in a short period of time have pulled sophisticated interaction techniques into most of them.

Building reliable interactive systems is a cumbersome task due to their very specific nature. Their behavior is usually event-driven making them belong to the reactive systems category. Beyond that, the main trigger for these events is the operator of the interactive systems usually behaving in an unexpected and unpredictable way. On the output side, information (e.g. the current state of the system) has to be presented to the operator in such a way as it can be perceived and interpreted correctly. Lastly, interactive systems require addressing simultaneously hardware and software aspects (e.g. input and output devices together with their device drivers).

Due to these specificities standard software engineering approaches cannot be reused for building reliable interactive systems. To address this challenge a lot of work has been carried out in the engineering interactive systems community extending and refining approaches including software architectures [7] formal description techniques and verification ([16], [8] and [14]) or testing ([12], [10] and [24]). Most of these works have been focusing on avoiding the occurrence of faults by removing software defects prior to operation i.e. during the development of the interactive system. The use of such techniques is particularly adequate when applied to safety-critical interactive systems (e.g. aircraft cockpits) as return on investment and cost-benefits trade-offs are covered by their safety-critical nature.

In the domain of fault-tolerant systems empirical studies have demonstrated (e.g. [23]) that software crashes may occur even though the development of the system has been extremely rigorous. One of the many sources of such crashes is called natural faults [3] triggered by alpha-particles from radioactive contaminants in the chips or neutron from cosmic radiation. A higher probability of occurrence of faults [28] concerns systems deployed in the high atmosphere (e.g. aircrafts) or in space (e.g. manned spacecraft [17]).

Such natural faults demonstrate the need to go beyond classical fault avoidance at development time (mainly based on formal description techniques and associated verification methods) and to embed fault-tolerant approaches to handle faults that may occur at operation time. In the area of dependable systems such issues have been studied and current state of the art in the field identifies four different ways to increase a system's reliability ([3] and [13]):

- *Fault avoidance*: preventing the occurrence of faults by construction (usually using formal description techniques and proving properties [25]).
- *Fault removal*: reducing the number of faults that can occur (by verification of properties).

These first two mechanisms belong to the so-called zero-defect approach aiming at acting in the development phase for preventing faults from occurring.

- *Fault forecasting*: estimating the number, future incidence and likely consequences of faults (usually by statistical evaluation).
- *Fault tolerance*: avoiding service failure in the presence of faults (usually by adding redundancy, multiple versions and voting mechanisms).

This paper focuses on the natural faults which will occur regardless the effort deployed during development phases. To increase the system reliability concerning these faults which occur during operations, this paper proposes a software architecture in order to address fault-tolerance. Fault-tolerance will be achieved by covering the following aspects:

- Fault detection: identifying the presence of faults, the type of the fault and possibly its source,
- Fault recovery: transforming the system state that contains one or more faults into a state without fault.

Beyond the proposed software architectures, we present how to use a model-based approach for describing, in a complete and unambiguous way, the various elements of the software architecture. Combining the zero-defect approach with the fault-tolerant one we will describe how to add fault detection and fault recovery mechanisms to interactive systems. This work thus extends previous work in the area of dependable computing by taking into account the specificities of interactive systems and adapting previous contribution to them.

As far as interaction technique is concerned, in this paper we are focusing on standard indirect manipulation techniques for which display and control take place through a predefined set of widgets (e.g. buttons, labels …). Even though a lot of more sophisticated interaction techniques are proposed and evaluated by the HCI community such indirect manipulation interaction style follows standards in the area of safety-critical interactive systems such as ARINC 661 specification for interactive civil cockpits [2].

Due to their standardized behavior and graphical representation user interfaces based on widgets are faster to design and easier to implement than the one offering direct manipulation interactions. This is the reason why user interfaces for critical command and control systems offer menu and form-based interactions based on standard widgets such as buttons, check boxes, radio boxes … Implementation of such interfaces usually exploit component based approaches [29] where the user interface consists in an assembly of reused software components.

Such approaches present a set of advantage improving reliability (as the components are reused and thus usually widely tested), development cost efficiency (components are produced by third parties and are uses "as is" by many "clients"), development time efficiency (the designer focuses only on the assembly of the components and not their design) …

This paper is structured as follows. Next section is dedicated to the generic architecture of component-based interactive systems. Section III proposes a software architecture for embedding fault-tolerance mechanisms in interactive systems and more precisely in widgets. To avoid faults in the design of the components enabling fault-tolerance, they have been formally modeled using a Petri nets-based formal description technique. In order to exemplify the concepts and to demonstrate their applicability, section IV presents their formal specification. The last section concludes the paper.

II. ARCHITECTURE OF A COMPONENT-BASED INTERACTIVE SYSTEM

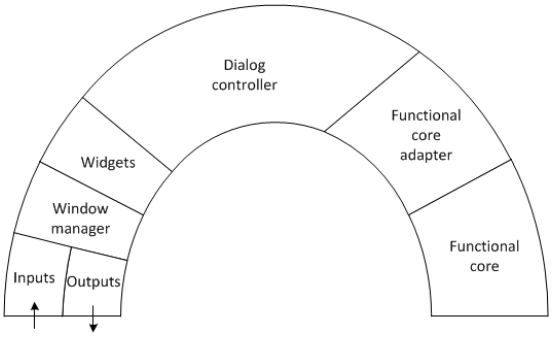

Figure 1. Architecture of critical interactive systems (from input devices to interactive applications) adapted from [7]

Figure 1 presents a revised version of the ARCH software architecture introduced in [9]. We have used it to explicit the functional architecture of most of the interactive and user-driven systems. Logical Level and Presentation Technique Interactive Components (the last two ones on the right-hand side of the original ARCH model) have been replaced by input and output devices, window manager and widgets. Indeed, from a functional point of view, a component-based interactive application (*Figure 1*, should be ridden from left to right) may be seen as a five parts system:

- *Input and output devices*: classically screens, keyboards, mice… ; but they can be more complex (e.g. combined devices such as the KCCU (Keyboard Cursor Control Unit) in interactive cockpits). They allow the interaction (at the hardware level) between the human and the computer.
- *Window manager*: embedding devices drivers, it manages the link between the input and output devices and the rest of the application. For example, it is responsible for the management of the graphical cursors, the identification of the widget which is targeted by user action on the input devices (called picking), the dispatching of the input device events to the corresponding widgets and the rendering of graphical information on the output device.
- *Widgets*: the basic interactive components. They are represented in a separate box in *Figure 1* as we consider here the widgets as independent components.
- *Dialog controller:* describes the application states and behavior and how events received from the previous components trigger state changes in the application and

how those state changes trigger rendering function execution in the lower level components.

- *Functional code adapter* and *functional core components* are embedding the non-interactive functionalities of the system.

III. AN ARCHITECTURE FOR FAULT-TOLERANT WIDGETS

A. Main Hypotheses and Functional Failures Taken into Account

This paper focuses on system-side dependability of interactive system and considers human-error as out of scope. This is indeed a very strong hypothesis but human reliability aspects can be considered independent from the ones addressed here and natural faults at operation time are not influenced by operator's behavior. As shown in section II these aspects are pertinent and are required to be dealt with adequately if the entire socio technical system is to be considered.

We only tackle the functional failures of the widgets ; the main part of the considered architecture in *Figure 1*. Moreover, input and output devices, dialog controller and functional core and the window manager are considered out of the scope of this paper.

Our proposed software architecture aims at ensuring that the interactive system processes correctly input events from operators, and renders correctly parameters received from the functional core. To be more concrete, we are targeting at managing three possible functional failures:

- **Erroneous display**: Incorrect display of data received from functional core (e.g. a widget receives a value to render and displays another value);
- **Erroneous control**: Transmission of a different action from the one done by the user (e.g. the user clicks on button1 but the application sends an event from button2);
- **Inadvertent control**: Transmission of an action without any user's action (e.g. an event click is sent to the application without user action on the input devices).

B. An Architecture for Fault-Tolerant Widgets

In this section, we describe a solution for building fault-tolerant interactive applications. To this end, we propose to apply and customize fault-tolerant architectures to the widgets.

Various architectures are available in the dependable computing community, each of them having drawbacks and advantages. We present in detail how to use a self-checking architecture to design widgets able to detect faults which occur during their execution.

1) An Architecture for Self-Checking Widgets

The architecture presented in Figure 2 is a good candidate for embedding fault-detection mechanisms into widgets. According to this architecture, the self-checking is made up of 5 connected sub-components:

- ***The façade*** is the envelope of the widget, coordinating the flow of data amongst 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 to the rest of the application (including other non-self-checking widgets). This is an important characteristic if one has to develop an applications embedding two types of widgets (a self-checking one and a non-self-checking one). As the self-checking mechanism requires a lot of resources (both at design time and at execution time), it should be used only for components involved in critical interactions, implying the coexistence of both fault-tolerant and non-fault-tolerant components within the same application.
- ***The dispatcher***: events and method calls received by the self-checking widget are forwarded to the ***dispatcher***. The ***dispatcher*** then duplicates events and sends them both to the ***functional*** and ***controller***. It manages all methods calls and events by means of a queuing mechanism: each input is stored and processed following the first-in/first-out principle. The ***dispatcher*** has to deal with temporal constraints of execution ensuring that ***functional*** and ***controller*** receive event in a synchronous way.
- ***The functional*** is the classical widget. The ***functional*** sends its outputs both to the self-checking widget and the ***comparator***.
- ***The controller*** is a second version of some of the functionalities of the widget. It only implements the functionalities that have to be supervised by the ***controller***. Its behavior is simpler than the one of the ***functional*** and is thus more reliable. The ***controller*** sends its output to the ***comparator***.
- ***The comparator*** is in charge of comparing the ***functional*** and ***controller*** outputs. There are two kinds of comparisons to perform: one related to parameters modification and the other related to event notification. When the ***comparator*** receives an output from the ***functional*** (resp. the ***controller***) it waits for the corresponding output from the ***controller*** (resp. the ***functional***). Following the reception of these two outputs, two types of errors can occur: one of the outputs is ill-timed (too late or too early) with respect to the defined temporal window or the outputs hold different values. In case of error, the ***comparator*** sends an error event) to the self-checking widget.

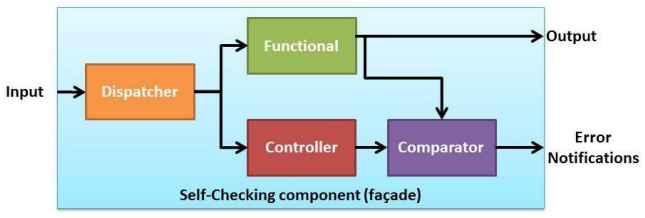

Figure 2. Self-Checking architecture for fault detection.

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 processor 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 [1] defines such partitioning in the domain of civil aviation and our contribution is compatible with that standard.

IV. Formal Specification of the Fault-Tolerant Interactive Components

It is clear that the proposed self-checking widget architecture relies heavily on the dependability of all its components. Indeed, all the components and mechanisms related to error detection must be reliable. Therefore, in a self-checking widget, the components enabling the fault-detection (e.g. the ***dispatcher***, the ***controller***, the ***comparator*** and the ***façade***) are supposed to be defect-free. We propose to ensure the integrity of the components enabling the fault-detection by the use of the ICO (Interactive Cooperative Objects) formal description technique to describe them in a complete and unambiguous way. In this section, we introduce a self-checking widget specification using the ICO formalism with the example of the self-checking PicturePushButton. We choose the example of the PicturePushButton as it is a widely used widget and is also representative of most interactive widgets defined in ARINC 661 as most interactive widgets share common properties such as visibility, enabling… Although this example might look rather simple at first glance, we can see that its behavioural description (presented on *Figure. 3*) is rather complex. All five components of the self-checking PicturePushButton (***façade***, ***dispatcher***, ***functional***,, ***controller*** and ***comparator***) are described using the ICO formalism.

A. ICO, a Formal description Technique

Interactive Cooperative Objects (ICO) is a formal description technique dedicated to the specification and verification of interactive systems [20]. 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 [19] to describe their dynamic or behavioural aspects. It is an extension of the Cooperative Objects formalisms that has been designed to describe behavioural aspects of objects-based distributed systems [9] and [10].

The formalism is able to handle the specific aspects of interactive systems. In a nutshell, the ICO formalism:

- Is Petri net based, suitable to specify the behavior of event driven-interactive systems and concurrent human-computer interactions and to describe the inner states of the Interactive Application.
- Enables the handling of more complex data structure (typed places and tokens, transitions with actions and preconditions, variable names on arcs).
- Allows objects of this type to react to external events according to their inner state and to produce events
- Defines an object as the set of four elements: a Cooperative Object which describes the behavior of the object, a presentation part, and two functions (the activation function and the rendering function) that make the link between the cooperative objet and the presentation part (events from input devices and output on the LCD screens).

B. ICO Modelling of the Self-Checking Widget

1) Classical Widget

In previous work [4], we have proposed the use of the ICO formal description technique for describing in a complete and unambiguous way standard widgets.

Figure. 3 shows the ICO model of a non-self-checking PPB (we only put here the snapshot of the model as the detail of its behavior is not interesting per se for the purpose of the paper). The parts of this behavior that are relevant for the fault-tolerant mechanism are explained when required in the following section but the entire behavioral description of the PPB can be found in [4].

The model of *Figure. 3* presents the various states the PPB can be in (e.g. visible, enable), the set of method calls he can process (e.g. processMouseClick, setLabelString), the set of events it can trigger (e.g. A661_EVT_SELECTION) and when such events are triggered (e.g. if the PPB is visible, enable and receives a method call processMouseClick the event A661_EVT_SELECTION is triggered).

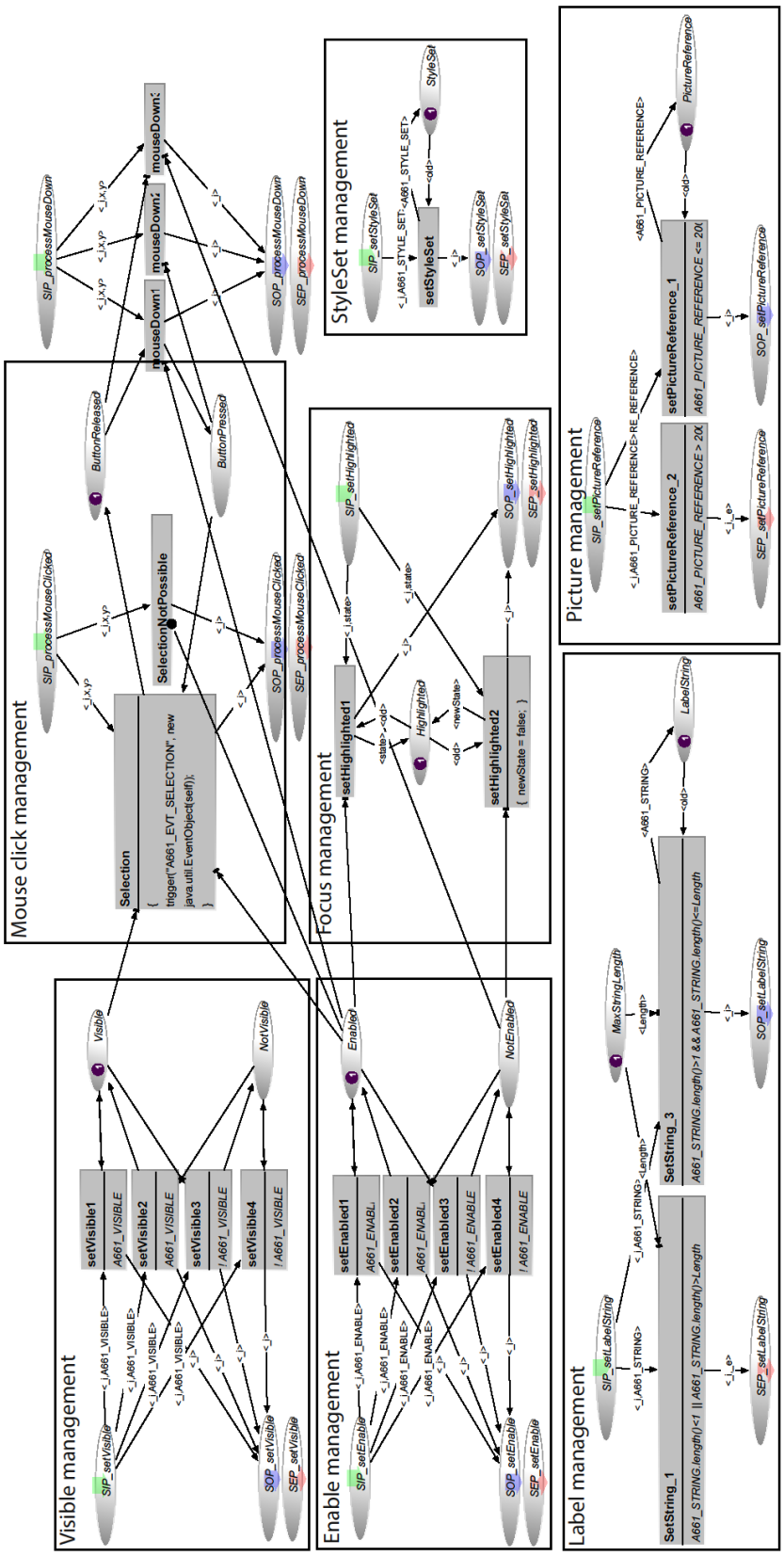

Figure. 3. ICO model of a PicturePushButton

2) Adding Fault-Tolerance

Adding fault-tolerance mechanism to the PicturePushButton is the result of the merge of the five subparts of the self-checking component architecture presented in *Figure 2* (the ***façade***, the ***dispatcher***, the ***functional***, the ***controller*** and the ***comparator***). As said previously, all five components of the self-checking PicturePushButton are described using the ICO formalism.

As explained in section III, the ***functional*** is the classical widget as illustrated by *Figure. 3* and the behavior of the ***controller*** must be close to the one of the ***functional*** as it has to provide the same outputs (events and parameters modifications) to be checked for conformity by the **comparator**. *Figure. 5* presents the ICO model of the ***controller*** of the self-checking PicturePushButton. We don't detail the behavior here but it is clearly different and simpler than the function described in *Figure. 3*. Thanks to the formal description performed using ICO, behavioral equivalence can be checked using results from the Petri nets theory such as the ones presented in [11]. These aspects are not relevant for the current paper but are of prime importance for the engineering of fault-tolerant interactive systems.

The ***façade*** being very simple to design as it is only a sort of wrapper allowing the application to discuss with the widget ; the main challenge is then to design the ***dispatcher*** and the ***comparator***. The ***dispatcher*** and the ***comparator*** of the self-checking PicturePushButton are two really large and complicated models, they cannot be presented in this paper due to lack of readability of the figure. However, the modeling of these two components has exhibited three generic patterns, one enabling the building of the ***dispatcher***; the others two enabling the building of the ***comparator***. We introduce here the specification of these generic patterns that can be seen as underlying building bricks of the components enabling the fault detection.

a) Dispatching Pattern

Figure 4 shows the pattern responsible for the dispatching of one input *setParameterX()*. To be able to communicate using a synchronous communication mechanism, the model needs a reference to the receivers i.e. the ***functional*** and the ***controller***. These references are stored in the places named resp. *functional* and *controller*. The input is received as a token in the place called *SIP_setParameterX*, and the value it holds is then associated to a queuing number (produced by the transition *setParameterX*) and stored in place *queueParameterX*. When the number of the next parameter to handle (this value is stored in place *nextToFireParameterX*) matches the value in place *queueParameterX*, the input value is sent to both ***functional*** and ***controller*** components by the firing transition *dispatchParameterX* and waits for confirmation. Finally, when the ***dispatcher*** has received both ***functional*** and ***controller*** acknowledgements (a token with the right queuing number must be held by places *ParameterX_F* and *ParameterX_C*), it is ready to process a new input, if any.

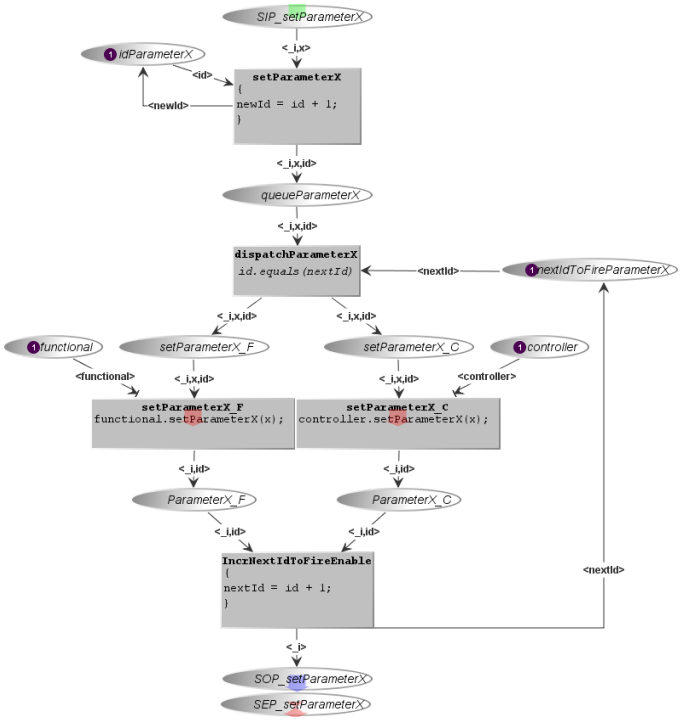

Figure 4. Dispatch of the input requesting the modification of the value of ParameterX of the PPB

This pattern is applicable to any kind of input services proposed by any widget (some small modifications may be needed on these patterns function of the service studied), for instance, the ***dispatcher*** of the self-checking PicturePushButton is composed of 14 of these patterns; one for each service proposed by the widget.

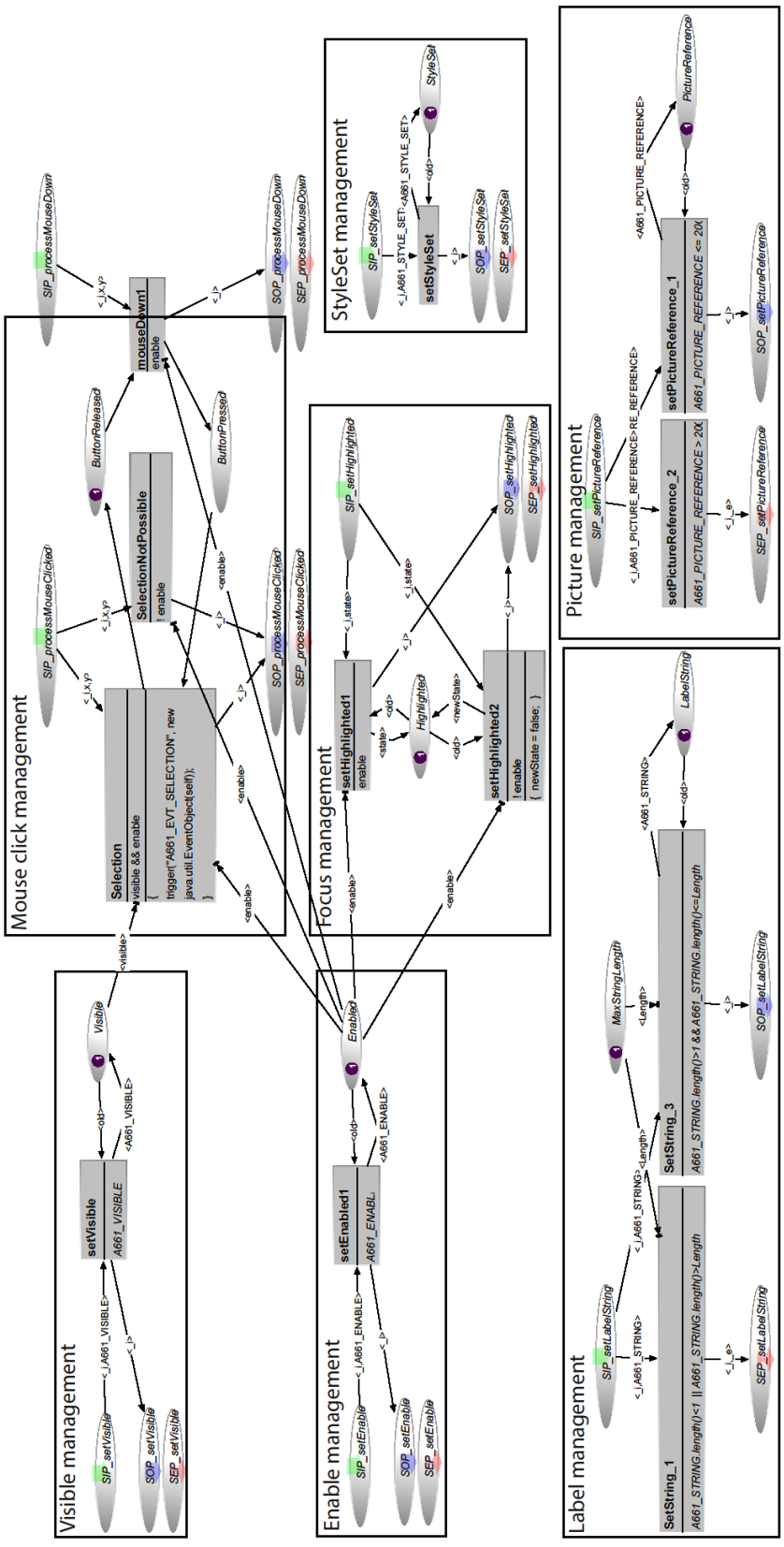

Figure. 5. ICO model of the controller of the self-checking PicturePushButton

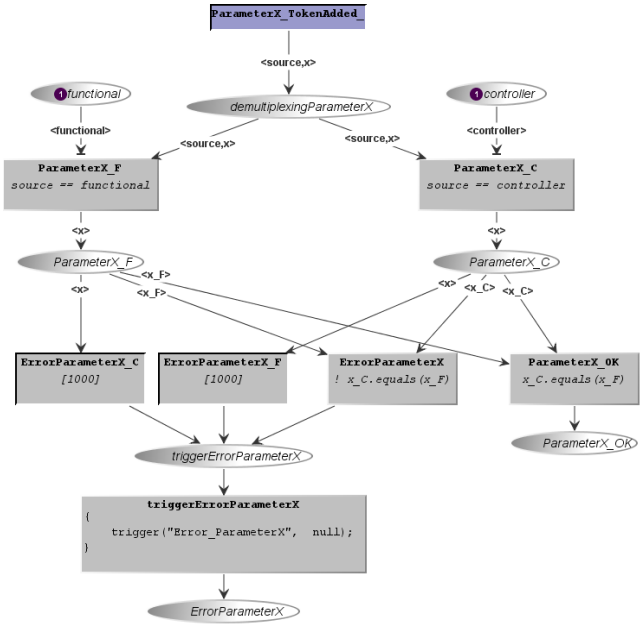

Figure 6. Comparison of ParameterX values

a) Comparator Patterns

We identified two generic patterns for the ***comparator***: (: (i) one for service execution (see *Figure 6*) ; (ii) one for event sending (see *Figure 7*).

Comparison of service execution

The pattern presented in *Figure 6* manages the comparison of the processing of a request of a parameter modification of the PicturePushButton. Four cases are possible and are represented by the four transitions at the center of *Figure 6*:

1. The value sent by the ***controller*** and the ***functional*** are the same, then transition *ParameterX_Ok* is fired and the ***comparator*** does not send an error notification (right-hand side of the figure).

2. The values received from both ***controller*** and ***functional*** are different, then transition *ErrorParameterX* is fired and the ***comparator*** raises as output an event "ErrorParameterX" that will be received by the application to which the widget belongs.

3/4. A more severe failure might occur in the ***controller*** or in the ***functional*** making one of them impossible to send a value to the ***comparator***. In such cases the corresponding timed transitions *ErrorParameterX_C* or *ErrorParameterX_F* is fired which results in triggering of the event "ErrorParameterX".

The case in which neither the ***functional*** nor the ***controller*** send a result is not considered here due to the segregation. Indeed, as both ***functional*** and ***controller*** have been developed independently and are executed in different partitions the probability of occurrence of such a case is below the 10^{-9} required for safety critical functions.

Comparison of event sending

The pattern presented in *Figure 7* exhibits a very similar behavior as the one presented in *Figure 6* and is applied to event processing. The only difference is related to the fact that the event does not carry any value and thus if it is received, its value is correct.

The ***comparator*** of the self-checking PicturePushButton is composed of 8 service execution comparison patterns and 1 event sending comparison pattern.

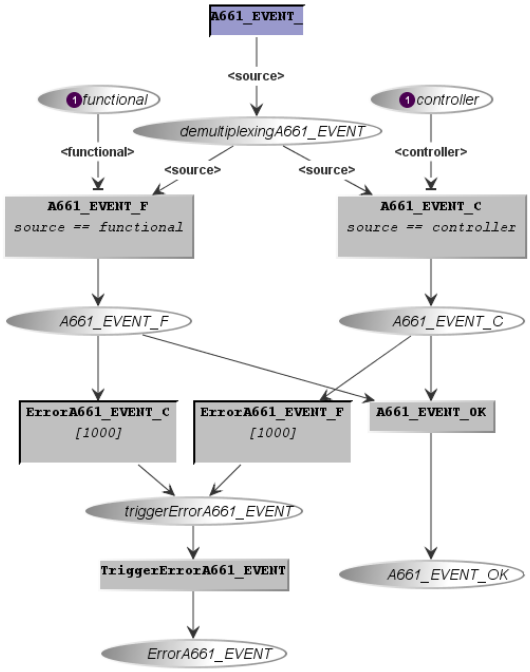

Figure 7. Comparison of an Event sending

V. CONCLUSION

In this paper we have presented that interactive applications in the context of safety critical systems (such as in interactive cockpits) raise specific issues about fault-tolerance. As interactions between the operator and the system takes place through standardized interactive components called widgets there is a need to enrich them with fault-tolerant mechanisms. We have proposed a new architecture and a model-based description of the components of an ARINC 661 widget that includes both a functional and a controller implementing runtime checks of the specified behavior of a widget. Despite the fact that the interaction techniques are rather limited (only WIMP ones) this contribution can be seen as a milestone for developing robust architectures for interactive cockpits.

The core of such an architecture (that includes such self-checking interactive component) is clearly its binding with the formal description technique thus providing a complete and unambiguous specification of such mechanisms.

This model-based approach is the backbone of a more ambitious project of providing notations, processes and tools for the engineering of dependable and fault-tolerancet

interactive applications. We are currently working on extending this self-checking mechanism to more components of the interactive system.

ACKNOWLEDGMENTS

This work is partly funded by Airbus under the contract CIFRE PBO D08028747-788/2008.

REFERENCES

[1] ARINC 653 Avionics Application Software Standard Interface. ARINC Specification 653. Airlines Electronic Engineering Committee July 15, 2003

[2] ARINC 661 Cockpit Display System Interfaces to User Systems. ARINC Specification 661. Airlines Electronic Engineering Committee 2002.

[3] Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. In IEEE Trans. on Dependable and Secure Computing, vol.1, no.1, pp. 11- 33, Jan.-March 2004

[4] Barboni, E., Conversy, S., Navarre, D., Palanque, P. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. DSVIS 2006. LNCS n°4323, pp. 25–38.

[5] Basnyat, S, Palanque, P, Schupp, B, Wright, P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design (2007) Safety Science, Vol 45, Issue 5, June 2007, ISSN: 0925-7535

[6] Basnyat S., Chozos N. and Palanque P. Multidisciplinary perspective on accident investigation. Reliability Engineering & System Safety Volume 91, n° 12, 2006, Pages 1502-1520

[7] Bass, L., Little, R., Pellegrino, R., Reed, S., Seacord, R., Sheppard, S., and Szezur, M. R. "The Arch Model: Seeheim Revisited." User Interface Developpers' Workshop. Version 1.0 (1991).

[8] Bastide, R., Palanque, P., Navarre, D., A Tool-Supported Design Framework for Safety Critical Interactive Systems, Interacting with computers, 2003, vol. 15/3, pp. 309–328.

[9] Bastide, R., Sy, O., Palanque, P. A formal notation and tool for the engineering of CORBA systems. Concurrency: practice and experience (Wiley) Vol. 12, pp. 1379-1403, 2000.

[10] Bastide, R., Sy, O., Palanque, P., Navarre, D. Formal specification of CORBA services: experience and lessons learned. ACM Conf. on Object-Oriented Prog. Sys. Lang. and Applications (OOPSLA'2000). ACM Press, p105-117.

[11] Bourguet-Rouger, A (1988). External Behavior Equivalence Between two Petri Nets. Concurrency 88, LNCS 335, Springer-Verlag, 1988, pp. 237-258

[12] Bowen J. and Reeves S.. 2011. UI-driven test-first development of interactive systems. In Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '11). ACM, New York, NY, USA, 165-174.

[13] Bowen J. and Stavridou V. Formal Methods, Safety-Critical Systems and Standards. Software Engineering Journal, 8(4):189–209, July 1993.

[14] Campos, J and Harrison, M. D. Formally verifying interactive systems: A review. 4th Eurographics workshop on Design, Specification and Verification of Interactive Systems DSVIS '97. 109-124. 1997. Springer Verlag.

[15] Dearden, A. M and Harrison, M. D. Formalising human error resistance and human error tolerance. Proceedings of the Fifth International Conference on Human-Machine Interaction and Artificial Intelligence in Aerospace. 1995. EURISCO

[16] Harrison M. & Dix A. A state model of direct manipulation. In M. Harrison and H. Thimbleby (eds.) Formal Methods in Human Computer Interactionpages 129-151, Cambridge University Press1990.

[17] Hecht H. and Fiorentino E. Reliability assessment of spacecraft electronics. In Annual Reliability and Maintainability Symp., pages 341–346. IEEE, 1987.

[18] Hollnagel, E. Barriers and Accident Prevention. 2004. Ashgage.

[19] Genrich, H.J. Predicate/Transitions Nets. High-Levels Petri Nets: Theory and Application, Jensen, K., Rozenberg, G. (eds.), pp. 3–43. Springer, Heidelberg (1991)

[20] Navarre, D., Palanque, P., Bastide, R. A Tool-Supported Design Framework for Safety Critical Interactive Systems, Interacting with computers, 2003, vol. 15/3, pp. 309–328.

[21] Navarre, D., Palanque, P., Ladry, J., and Barboni, E. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability, ACM TOCHI, 2009, V. 16, 4, pp. 1-56

[22] Neema S., Bapty T., Shetty S. & Nordstrom S. 2004. Autonomic fault mitigation in embedded systems. Eng. Appl. Artif. Intell. 17, 7 (October 2004), 711-725.

[23] Nicolescu B., Peronnard P., Velazco R., and Savaria Y. 2003. Efficiency of Transient Bit-Flips Detection by Software Means: A Complete Study. In Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '03). IEEE Computer Society, Washington, DC, USA, 377-384.

[24] Palanque P., Barboni E., Martinie C., Navarre D., and Winckler M. 2011. A model-based approach for supporting engineering usability evaluation of interaction techniques. In Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '11). ACM, New York, NY, USA, 21-30

[25] Pnueli A Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. LNCS n° 224 p.510-584. Springer Verlag 1986.

[26] Polet, P, Vanderhaegen, F, and Wieringa, P. Theory of safety related violation of system barriers. Cognition Technology & Work, 4, 3, 171-179. 2002.

[27] Reason, J. (1990). Human Error, Cambridge University Press

[28] Schroeder B., E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In ACM SIGMETRICS, pages 193–204, Seattle, WA, June 2009.

[29] Szyperski C., Gruntz D. & Murer S. Component Software - Beyond Object-Oriented Programming. 2nd Edition Addison-Wesley / ACM Press, 2002 (608 pages) ISBN 0-201-74572-0