A Development Process for the Design, Implementation and Code Generation of Fault Tolerant Reconfigurable Real Time Systems

Wafa Gabsi

ReDCAD Laboratory,
National School of Engineers of Sfax,
University of Sfax,
B.P. 1173, 3038 Sfax, Tunisia
E-mail: wafa.gabsi@redcad.org

Bechir Zalila

ReDCAD Laboratory,
National School of Engineers of Sfax,
University of Sfax,
B.P. 1173, 3038 Sfax, Tunisia
E-mail: bechir.zalila@enis.rnu.tn

Jérôme Hugues

MARS Research Group,
Institut Supérieur de l'Aéronautique et de l'Espace,
Université de Toulouse, France
E-mail: jerome.hugues@isae.fr

Abstract: The implementation of hard real-time systems is extremely a hard task today due to safety and dynamic reconfiguration requirements. For that, whatever the taken precautions, the occurrence of faults in such systems is sometimes unavoidable. So, developers have to take into account the presence of faults since the design level. In this context, we notice the need of techniques ensuring the dependability of real-time distributed dynamically reconfigurable systems. We focus on fault-tolerance, that means avoiding service failures in the presence of faults. In this paper, we have defined a development process for modeling and generating fault tolerance code for real-time systems using aspect oriented programming. First, we integrate fault tolerance elements since the modeling step of a system in order to take advantage of features of analysis, proof and verification possible at this stage using AADL and its annex Error Model Annex. Second, we extend an aspect oriented language and adapt it to respect real-time requirements. Finally, we define a code generation process for both functional preoccupations and cross-cutting ones like fault tolerance and we propose an extension of an existent middleware. To validate our contribution, we use AADL and its annexes to design a landing gear system as an embedded distributed one.

Keywords: Fault-tolerance, modeling, aspect-oriented programming, real-time, dynamic reconfiguration, AADL.

Reference to this paper should be made as follows: xxxx

Biographical notes:

Wafa Gabsi is a PhD student within the ReDCAD Laboratory at the National School of Engineers, at the University of Sfax, Tunisia. She received her Master's in Computer Science at Sfax University in 2011. Her study involves several proposals destined for taking into account the verification, the analysis and the proof of fault tolerant dynamically reconfigurable real time systems since the modeling step. She works in the context of a collaboration between the National School of Engineers of Sfax (ENIS) and the Institute for Space and Aeronautics Engineering (ISAE).

Bechir Zalila is an associate professor at the Department of Computer Science Engineering and Applied Mathematics of the National School of Engineers of Sfax, Tunisia (ENIS). He had a PhD in computer science in 2008 and an engineering degree in 2005, both from Telecom ParisTech. His research within the ReDCAD Laboratory focus on design, analysis and implementation of reconfigurable real-time, embedded and distributed systems.

Jérôme Hugues is an associate professor at the Department of Mathematics, Computer Science, and Control of the Institute for Space and Aeronautics Engineering (ISAE). He held his PhD in 2005 and his engineering degree in 2002 from Telecom ParisTech. His research interests focus on design of software-based real-time and embedded systems and tools to support it. He is a member of the SAE AS-2C committee working on the AADL and is involved in the Ocarina and TASTE projects.

1 Introduction

A quick evolution of real-time distributed dynamically reconfigurable systems is noticed nowadays following the emergence of new needs in this area. In such types of systems, whatever the taken precautions, errors occurrence cannot be avoided due to internal or external conditions like human-made faults, equipment aging, natural disasters, etc. In some cases, faults can lead to serious consequences such as loss of money, time or even lives.

For these reasons, these systems must introduce high dependability [1, 2], which is defined as the ability to deliver a service that can justifiably be trusted. This concept is surrounded by various attributes that are: availability, reliability, safety, integrity and maintainability. All these properties must be satisfied even in the presence of dependability threats which are fault, error and failure. The means that we usually need to avoid different types of faults include fault prevention, fault forecasting, fault removal and fault tolerance. Fault Tolerance (FT), one of the different means of dependability, is defined as the capability of a system to continue providing offered services even in the presence of errors [1]. Based on redundancy to detect and mask errors, fault tolerance can reveal major problems in the context of real-time dynamically reconfigurable systems.

First, software fault tolerance requires dynamic reconfiguration tasks at runtime. This can compromise the execution predictability. In the context of real-time systems, this will

be a source of non-determinism. Second, fault tolerance mechanisms are based on some redundancy. They introduce additional complexity and cost to the core functionality. So, we have to verify that fault tolerance techniques are feasible from an overhead perspective which must be acceptable in terms of execution time and memory footprint. Reducing the risk increases the overhead. In order to provide powerful redundancy at lower cost, we must define the fault tolerance mechanisms appropriate to the system. In addition, in the context of dynamically reconfigurable systems, we must take into account new considerations related to the migration between configurations. Adaptation, in this case, may run the risk of a failure occurrence during a reconfiguration. Besides, there is a lack of techniques allowing real-time fault tolerant code generation from dependable model. They only aim at formally verifying certain properties through model transformation.

To overcome these problems, we present in this paper our work devoted to support fault-tolerance for real-time distributed and dynamically reconfigurable systems. For these types of applications, we integrate the fault tolerance elements in the modeling of a system to benefit of analysis, proof and verification features feasible at this stage. The main purpose of this approach is to enhance the fault coverage ensured by FT strategies dedicated to some selected classes of faults at design level as well as the implementation level. Since fault tolerance is a cross-cutting preoccupation, we decided to implement the various algorithms using aspect-oriented programming (AOP) to be coded separately.

For that, we define a development process for the design, the implementation and the code generation of fault tolerant reconfigurable real time systems. We use AADL to model our core system. Then, we offer the opportinuty to extend this model with fault tolerance aspects using AADL annexes such as Error Model Annex and Behavioral Annex. We propose then the code generation for both functional and cross-cutting concerns using different tools to have finally fault tolerant code. We chose to generate application code into Ada language because it is well adapted to implement real-time embedded systems. For compliance reasons, we generate aspects into AspectAda. In order to validate our approach, we model a landing gear system using AADL and its annexes as a case study.

The remainder of this paper is structured as follows: In Section 2, we present background concepts related to dependability threats, fault classification and fault tolerance techniques. Section 3 detailes the related work. In Section 4, we introduce our approach through a detailed production process description. Then, in Section 5, we illustrate our contribution by applying it within an example of an embedded distributed systems. Finally, Section 6 concludes this paper and gives future work.

2 Background

This section introduces the basic concepts of dependability threats, fault classification and fault-tolerance techniques.

2.1 Dependability threats

All the means of dependability address similar threats that are faults, errors and failures [2].

- **Fault**: physical hardware or software defect; any event, action or circumstance causing service degradation.
- **Error**: incorrect value causing a system failure.

- **Failure**: deviation of the system relatively to specification.

The error propagation is a major risk for dependability. An error is the manifestation of the fault on the system. A service failure occurs when an error is propagated to the service interface and deviates the service from its correct specification. Moreover, given that a computer system is often composed of several subsystems, a failure of a subsystem can cause a fault in an another subsystem or in the system itself.

2.2 Fault classification

Faults are classified according to eight basic viewpoints, leading to the elementary fault classes [2].

Criteria	Types	Description
Objective	Malicious	introduced by a human with an objective of causing harm to the system
	Non-Malicious	introduced without malicious objectives
System boundaries	Internal	originating from inside the system
	External	originating from outside the system
Phenomenological cause	Natural	caused by a natural phenomena without human participation
	Human-made	resulting from human actions
Phase of creation or occurrence	Conceptual	occurring at design level
	Operational	occurring at execution level
Persistence	Permanent	assumed to exist continuously
	Transient	whose presence is bounded in time
Dimension	software	originating or effecting on software components (programs and data)
	hardware	originating or effecting on hardware components

Table 1 Faut Classification

In our context, we focus on a particular set of faults composed of six elementary fault classes as it is shown in Table 1.

2.3 Fault tolerance techniques

In order to avoid the system failure, fault tolerance is accomplished through the following techniques:

1. **Error detection :** consists on detecting error occurrence. It is either concomitant or preemptive.
   - Concomitant detection, carried out during the regular execution of the system.
   - Preemptive detection, achieved while a regular execution of the service is suspended.

2. **System recovery :** consists in replacing the erroneous state of the system by another safe state. This phase is based on two mechanisms:
   - Fault handling, prevents the activation of fault once more. This can be achieved by various ways:
     - *Diagnosis*: records and identifies the cause of the error in terms of type and location.
     - *Isolation*: deactivates the fault by ensuring physical or logical exclusion of faulty components.
     - *Reconfiguration*: assigns tasks to non-faulty components.
     - *Reinitialization*: checks, updates and records new configuration and data of the system.
   - Error handling, eliminates errors from the state of the system. In this case, we have to use rollback, rollforward or compensation or combine some of them in a particular situations.
     - *Rollback recovery*: turns up the erroneous state of the system to an earlier steady state saved.
     - *Rollforward recovery*: corrects the system state by moving it to a new steady state.
     - *Compensation*: to enable masking the erroneous state, the system provides enough redundancy from the beginning. **Redundancy** is a practice commonly used in the design of critical or fault tolerant systems. It involves repetition and multiplicity of different components or treatments.

3 Related Work

This section gives an overview of research work related to modeling fault tolerant mechanisms. Then, it introduces some work using aspect-oriented programming for fault tolerance purposes as they consider fault tolerance as a cross-cutting concern.

3.1 Design of fault tolerance

In [3], authors offered the modeling of certain types of faults uniformly using transition systems in order to apply model checking and model revision techniques effectively. In each of 31 categories in the taxonomy [1] of faults, they focus on how faults can be modeled and whether it is feasible using transition systems. After studying their one by one, they concluded that such modeling is feasible and not practical for two categories. Moreover, it is not feasible for 11 categories. These 13 categories include the all possible combination of internal faults. They present a significant increase in reacheable states and they need the design of continuous behavior. The remaining 18 categories, for which this approach is feasible and practical, are all external faults. So we cannot use such model technique to design fault tolerance of distributed embedded systems.

Other viewpoints focused on modeling Fault Tolerant techniques using ADLs (Architecture and Description Languages) like AADL and others on model driven approach to define a fault tolerant development process for such systems. In [4] and [5], the authors extended the framework FT-CORBA with support of fault-tolerance mechanisms such as

fault detection and redundancy. Thus, they defined a modeling process in order to be able to design fault tolerance components and generate their correspondent code. This modeling process is based on UML with CCM (*CORBA Component Model*) as well as the QoS& FT (*Quality of Service and Fault Tolerance*) profile. After that, this model can be deployed and executed by means of CCM descriptor files. Yet, this approach does not support the dynamic reconfiguration of fault tolerance components. Also, authors did not evaluate the implementation of their framework with different fault tolerance strategies. They limited its test to a unique replication style that is the active one with voting.

In [6], authors used AADL to model high-integrity systems. They also used AADL Behavioral Annex which extends AADL with the refinement of behavioral aspects expressed as a state transition automaton with guards and actions using variables to manipulate data. They integrated fault tolerance concerns by applying replication patterns in order to implement distributed real-time embedded systems. To express the modeling capabilities of AADL and its annex, the authors considered a simplified design of the PBR (Primary Backup Replication) pattern. They successfully designed the PBR strategy with the integration of a set of very strict semantic rules that are consistency with the core language at the architectural level. However, this Behavioral annex is not sophisticated to the design of fault tolerant components. It allows it implicitly without specifying error types or error behaviors unlike the Error Model Annex.

The authors of [7], defined a new dependable AADL model which describes the architecture of the system and its dependability information separately using respectively AADL and its annex Error Model Annex. In this context, they defined a modeling process based on model transformation in order to design dependable and secure systems. Then, they annotate the architectural model with Error Model Annex constructs. After the definition of model transformation rules, the authors allow the generation of GSPN (General Stochastic Petri Nets) from AADL models to validate it against its specification, based on the analysis and validation of the GSPN model, after each iteration. Nevertheless, this approach does not offer the code generation of fault tolerance mechanisms. It only offers the verification of the AADL dependability model as well as the previous approaches.

3.2 Fault tolerance using AOP

There are several work which offer fault tolerance for computing systems. We focus on a subset that used aspect-oriented programming to implement their different algorithms.

Authors in [8] proposed a fault injection study that estimates the fault coverage of two software implemented with AOP. They investigated the Double Signature Control Flow Checking (DSCFC) and the Triple Time Redundant Execution with Forward Recovery (TTRFR) mechanisms. They also implemented them using three different languages to prove the importance of AOP languages to reduce the overhead. However, this approach is not adapted to dynamically reconfigurable systems. Besides, it presents the problem of glue code which is generated by the AOP language weaver and not visible in the program sources. So, it might be vulnerable to faults.

Other fault tolerant startegies were implemented and integrated into a framework for distributed embedded fault tolerant systems based on AOP in [9]. There are : the Recovery Blocks (RB), the Distributed Recovery Blocks (DRB) and the N-Version Programming (NVP). The second strategy is the distributed nature of the first one which is based on rollback recovery. While the third strategy is based on masking errors and the majority voting to select the correct response. This approach is promoted among the previous one as

it is adapted not only to real-time systems but also to distributed dynamically reconfigurable ones thanks to transparently communication between nodes based on Publish/Subscribe protocol.

In [10], the authors proposed an aspect oriented framework for errors detection, recovery mechanisms and exception handling design framework based on plausibility checks. The aspect oriented design patterns brought by this framework offer additional benefits like the localization of error handling code in terms of definition, initialization and implementation. This approach proves re-use and portability of its framework thanks to the separately implementation of the error treatment with fault injection of permanent faults. Nevertheless, this approach is not adapted neither to embedded nor real-time systems. Indeed, it uses best effort sensors for detecting failures.

Finally, the authors of [11], proposed using the AOP in order to improve performance of the fault tolerance CORBA services (*FT-CORBA*). They ensure quantitative evaluation of the performance gains which asserts the overhead decreasing at runtime comparing it to other implementations studied. This approach offers a separate implementation of both journalization and synchronization mechanisms. It also brings an optimization of the interceptors in the FT-CORBA infrastructure to exception handling at the application level. However, like the previous one, this approach is not adapted to neither real-time nor embedded systems since the FT-CORBA middleware presents high print memory costs. In addition, integrating aspects into the FT-CORBA infrastructure can involve decreasing the security level and maintainability difficulty. Besides, this approach with all previous ones, do not offer modeling fault tolerance strategies.

The main difference between the reviewed approaches above and our approach is the focus on: (i) its adaptability to real-time distributed and dynamically reconfigurable systems on one hand and (ii) its modeling and code generation of fault tolerance requirements on the other hand. That is they integrate the Fault Tolerance preoccupations at the development level using AOP. Also they do not take into consideration the effect of weaving on the system scheduling and its determinism.

In our work, we aim at extending these work by modeling fault tolerance elements with respect of real-time constraints and fault tolerance policies. Compared to the UML QOS and Fault Tolerance profiles, which is limited to FT-CORBA architectures, our work offers to the user the liberty to design different types of errors and to model software and hardware architectures using AADL and its Error Model Annex and specify semantic behaviors using the Behavioral Annex. It also offers the dynamic reconfiguration policies thanks to the AADL specification of modes and mode transitions.

4 Proposed Approach

Aiming at ensuring fault tolerance for distributed real-time dynamically reconfigurable systems, we defined a development process which integrates fault tolerance mechanisms since the modeling stage.

First of all, we selected a subset of the studied software errors that we can deal with in the context of real time dynamically reconfigurable systems, as it is shown in Figure 1 (Faults with text in bold are those we can handle for these types of systems).

Since the malicious faults are related to security and then to the user intention, we consider non-malicious faults. We focus on internal and natural faults as we consider embedded dynamically reconfigurable systems. We consider also conceptual faults as we

aim to integrate fault tolerance elements since the design level. Besides, as we design fault tolerance for real-time systems, the service delivery must be on time. So, we focus on the two persistence types of faults, permanent and transient ones. In our case, we do not handle byzantine faults because in real-time critical systems, we will incur the risk if we consider such errors .

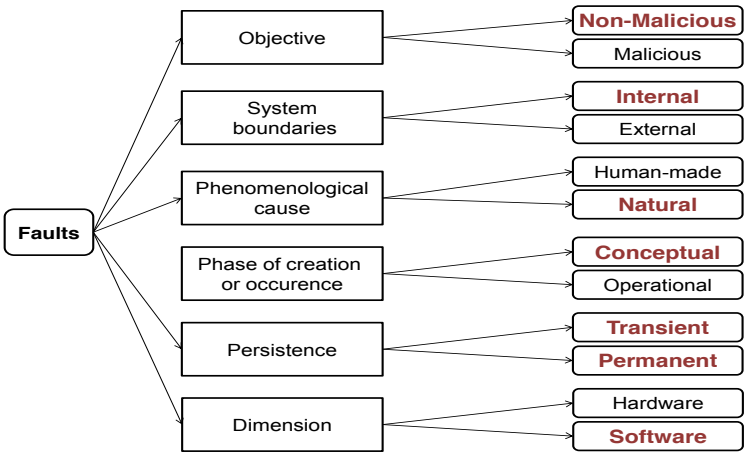

Figure 1: The fault classification

Then, we selected AADL [12] (*Architecture Analysis & Design Language*) as an Architecture Description Language to integrate fault tolerance elements at design level for these reasons.

AADL is a standard consisting of both a textual and graphical representations with precise execution semantics for embedded software systems. In addition, AADL is a concrete language: all components of an AADL model correspond to concrete entities. Also, this language can model an entire system including both software and hardware components.

Besides, AADL provides a support to describe operational modes of a system and then manages dynamic reconfiguration. Modes and mode transitions describe the reconfiguration processes of existing components. A mode represents an operational state, which manifests itself as a configuration of contained components, connections, and mode-specific property value associations. But a configuration, is statically defined consisting of a set of components linked with connections. AADL can also be used to describe the dynamic behavior of the runtime architecture due to its modes and mode transitions concepts. So, the switching between configurations consists on transition between modes. That needs only the specification of reconfiguration constraints to guarantee that mode transition brings us to a novel safety mode. Moreover, AADL can be extended with properties to specify critical characteristics of a component within its architectural context using properties. AADL annex libraries enable a designer to extend and customize the AADL core specification with other concepts specified in a language other than AADL.

Looking for the adequate fault tolerance strategies and the best modeling techniques to design the selected faults, we decided to use the AADL standard annex, Architecture Analysis & Design Language (AADL) Annex E: Error Model Annex [13]. This annex, aims to model not only different types of faults, but also fault propagation affecting related

components. In fact, this annex lets us to design all types of faults, fault behavior, fault propagation, fault detection and also fault recovery mechanisms. The language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. To extend this language and completely describe the system, we can add new properties and analysis specific notations that can be associated with components.

Based on AADL and its Error Model Annex, we proposed our approach offering a development process integrating fault tolerance requirements at modeling stage as it is shown in Figure 2.

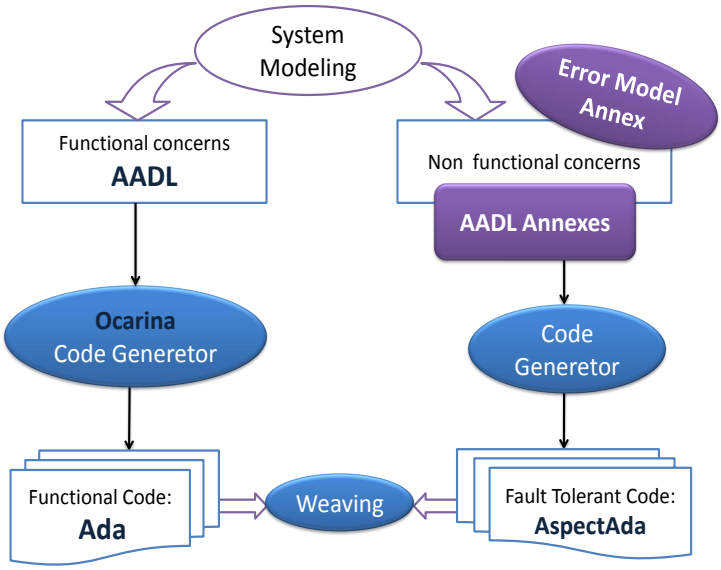

Figure 2: The proposed development process

We use AADL to model functional as well as non-functional concerns. In particular we distinguish fault tolerance requirements which we design using the Error Model Annex. For that, the designer can use the OSATE [14] tool which offers a collection of meta model packages with a textual and an XMI specification from Ecore meta model. This tool lets the designer modeling its core system with adding properties and fault tolerant specification by using Error Model Annex.

Once the model is designed using the AADL language, an automatic generation code process used to obtain an adequate code using Ocarina tool suite [15]. It allows to generate the code corresponding to the functional part of an AADL specification into Ada, C, or RealTimeJava. Then, software embedded systems can be deployed on several platforms. We use Ocarina to generate the functional code in Ada [16] language, for these reasons:

- Ada is adapted to real-time embedded critical systems. Some Ada execution profiles help to have systems statically analyzable and can enable the contracts to be proved prior to execution. For instance, the Ravenscar [17] profile is a subset of Ada tasking

intended to model concurrency in safety critical, high-integrity, and general real-time systems by imposing restrictions at compile-time using Ada pragma restrictions.

- Ada can be combined with various formal proof technologies to formally prove certain properties of the code. SPARK [18], an annotation system for Ada, provides assertions regarding state in order to ensure dynamic proof and checks.

As we have already mentioned, we decided to implement cross-cutting concerns seperately with aspect oriented programming [19]. For compliance reasons, the fault tolerant code as well as non functional concerns code have to be generated in AspectAda [20] in order to be weaved automatically into the functional code generated in Ada. In the context of real-time systems, the weaving operation can compromise determinism. Therefore, it has to avoid several constructs such as dynamic allocation, dynamic priorities and dynamic dispatching. These constructs make the estimation of execution time hardly predictable. For that, we studied the existing language and extracted its limits and deficiencies then we proposed the redevelopment of the language from scratch [21]. On one hand, the prototype implementation of an AspectAda compiler is still in an early stage. The compiler cannot yet deal with all AspectAda language constructs in all possible contexts. It was also non specific and poorly structured. On the other hand, this language, as it is published, introduces a set of gaps. Its grammar has several deficiencies and its runtime is very poor. To remedy these shortcomings, we optimized and extended the existing AspectAda language to adapt it to real-time requirements.

We started by extending AspectAda to export the context of joinpoints, arguments and returned value, to the advice code and its grammar to include more concepts of the aspect oriented paradigm like both types of joinpoint `call` and `execution`. Then we defined the weaving and generation code rules based on real-time requirements. After that, we have developed the aspect and weaver APIs and extending the aspectAda runtime. Then, we defined a novel architecture of the language compiler similar to the architecture of modern compilers as shown in Figure 3. The new compiler AspectAda architecture consists of three parts:

- The central library defines a set of routines useful for construction and manipulation of syntactic trees (AST).
- The frontend allows lexical, syntactic and semantic analysis of three types of source code: the aspects, the weaver and the functional Ada code. Each front-end part generates as output a decorated AST using routines of the central library.
- The bakcend receives three ASTs resulting from the front-end. Then it crosses and interviews different ASTs to produce a single Ada AST from which the Ada woven code will be generated.

From the implementation viewpoint, we accomplished the development of the three units: Ada unit, the Aspect unit and the weaver unit each itself composed of lexer, parser and semantic analyser and generates as output a syntactic tree. These trees are the input of the backend part that we are actually developing to implement the weaving and the woven code generation process.

As we have already mentioned, we will generate fault tolerant code corresponding to the "Error Model Annex" in AspectAda to be woven with functional code generated in Ada. For that, we are actually defining the code generation rules compliant with real time systems

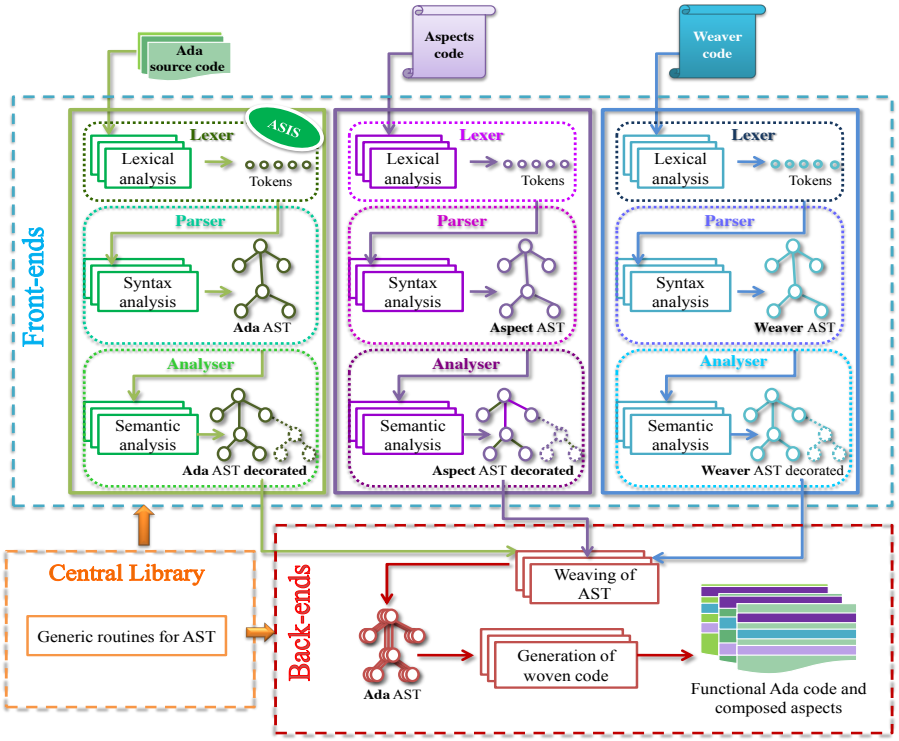

Figure 3: The proposed architecture for the AspectAda compiler

by avoiding constructions may be harmful to determinism like dynamic allocation. After that, we aim at implementing the automatic code generator while applying these defined rules [21].

In addition, building a middleware for distributed real time system is a tadious task. In particular, for fault tolerant systems, it will be even a harder task. In our context, we aim at generating a part of the application code , as large as possible, automatically from our AADL verifiable and analysable model. For that, we are actually applying our own approach in order to extend the PolyORB High Integrity middleware [22] to be conform to many functional and non-functional requirements especially fault tolerance ones to describe the critical aspects. We are identifying the middleware required properties and functionnalities to efficiently implement the Fault Tolerant specification. We aim at extending and adapting the Polyorb_HI by generic components designed to provide application level entities with application Programming Interfaces (APIs) to run fault tolerance requirements. Our purpose is a complete design and configuration of the support provided by the middleware for the fault detector, fault recovery, replication and consensus components. Thus, we not only reduce the development costs but also we make easier the work of the developer and help in code certification.

In this section, we have defined our own approach that offers fault tolerance from modeling up to code generation with respect of real-time constraints. Based on our development process, we offer the designer the ability to model functional concerns with

AADL as well as cross-cutting ones with annexes. In particular, we integrate the fault tolerance elements corresponding to some classes of faults already selected in the model using AADL Error Model Annex. The user can also, if he want, use the AADL Behavioral annex to specify semantics behaviors of core or fault tolerant components or timing constraints. Then, we choose to generate functional code in Ada and non-functional code in AspectAda which we adapted and extended to real-time constraints.

5 Case Study: Landing Gear System

To validate our contributions proposed in Section 4, we chose the Landing Gear of an aircraft system [23] as a case study. This case study is used and implemented in the context of a collaboration between the National School of Engineer of Sfax (ENIS) and the Higher Institute of Aeronautics and Space (ISAE). Through this case study, we look at giving an example of a critical real time embedded system applying our approach to model functional and fault tolerance concerns. Considering such system, we aim firstly at modeling and programming the software part controlling the landing and the retraction sequence, and secondly at detecting and recovering if possible anomalies on hydraulic equipment, electrical components, or computing modules taking into account the physical behavior of hydraulic devices. The action to be done to apply one of the two basic scenarios, the outgoing sequence or the retraction one, depends on the state of all the physical equipements and on their temporal behavior at each time.

To model our system, we used the open-source tool platform OSATE2 in order to verify the lexical, syntactic and semantic analysis of our model. It provides not only a full support for the AADL meta-model on an Eclipse platform to developers but also a complete textual editor for AADL to the users. Besides, OSATE2 have an official set of plug-ins including the Instance Model Viewer (IMV) and the the Error Model Annex V2. The former lets the user instantiate his model first and then shows the graphical architecture. The latter ensures different analysis of the dependable model in particular Fault Impact, FHA and Reliability Block Diagram (RDB). Then, to generate functional code from the core AADL Specification in Ada, we used the Ocarina tool set.

First, we describe the architecture and the functionalities of the landing gear system that we used AADL to model its components and connections. Then, we detail the extension of the AADL model by annexes to describe cross-cutting concerns in particular fault tolerance using AADL Error Model Annex and the consensus functionalities using AADL Behavioral Annex.

5.1 Description

This case study was proposed as a benchmark for techniques and tools dedicated to the verification of behavioral properties of systems. Composed of 3 landing sets (front, left and right), the landing system is in charge of maneuvering landing gears and associated doors. In fact, each landing set contains a landing-gear, a door and associated actuating hydraulic cylinders. To facilitate the charge of the pilot when controlling this system, its is composed of three parts like it is presented in Figure 4:

- A mechanical and hydraulic part which contains all the mechanical devices and the three landing sets. In particular, the mechanical part has actuating cylinders responsible on one hand for opening or closing doors and on the other hand for retracting or

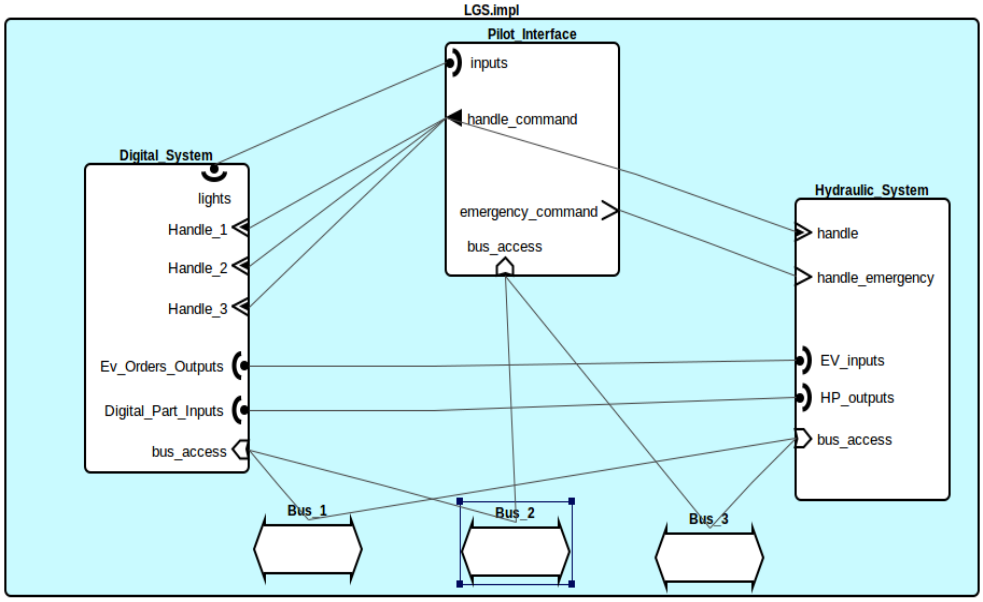

Figure 4: The AADL global architecture of the landing system

extending gears. Hydraulic power is provided to the cylinders by a set of electro-valves. Each of them is activated by an electrical order coming from the digital part. Each of these mechanical equipments (Doors, gears, cylinders and electro-valves) is associated with a set of discrete sensors to inform the digital part about its state. For the purpose of preventing sensor failures, each sensor is triplicated. So, it deliveres simultaneously three discrete values describing the same situation.

- A digital part including the control software is executed simultaneously by two identical computing modules. In order to monitor mechanical devices, detecting anomalies and informing the pilot about the system state, these modules produce commands for the hydraulic part and give information about the states of the system and its different equipements to the pilot. To do so, the two computing modules receive the same triplicated inputs and select one of these three values by applying a consensus algorithm as shown in Figure 5.

- A pilot interface provides the "UP/Down" handle in order to command the retraction and outgoing of gears. It lets also the pilot monitoring the states of the system and its different equipements as well as the current position of gears and doors through a set of lights. In case of failure, the pilot can manually activate the emergency hydraulic circuit.

Since we apply our approach to a real time system, it is compulsory to verify the scheduling properties of the system. For that we used the Cheddar tool [24, 25] which is designed for checking task temporal constraints of a real time system described with AADL. From another perspective, we enriched the AADL core system specification by pieces of annexes to describe the fault tolerance requirements that we detail in the next Section.

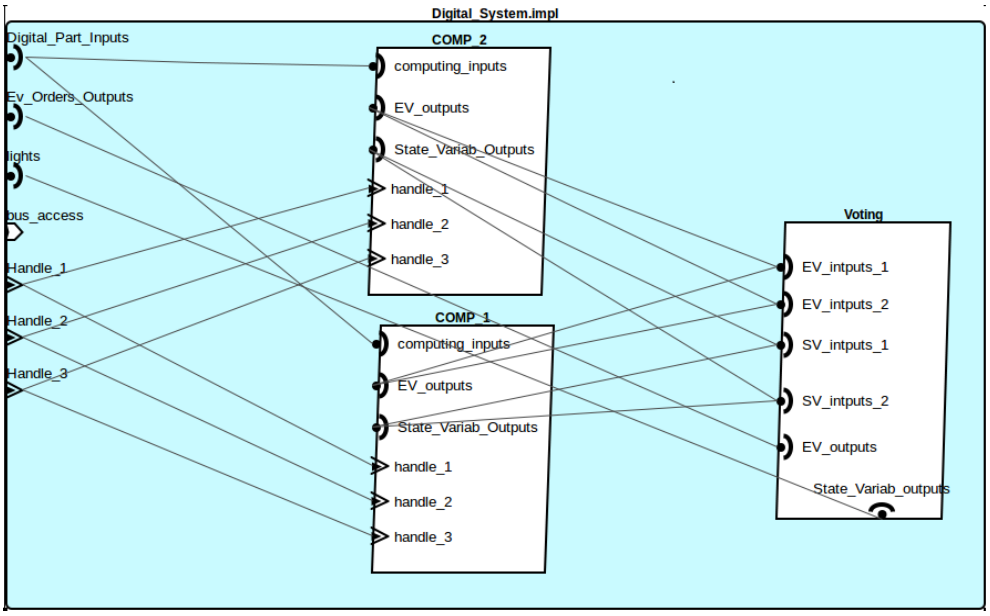

Figure 5: AADL Specification of the Digital Part

5.2 Modeling the fault tolerance requirements

After modeling the core AADL specification of our system, we extracted all types of faults that the system can occur, the possible propagations and the different behaviors prevented for each of them according to the system specification. We give in this Section some examples of "Error Model Annex" sequences coded at model level to describe error behaviors of some components.

Thus, we began by defining a package policies that contains a set of prevented errors. The Listing 1 describes this package.

Listing 1: AADL Specification of the possible errors, task

```
1  package Error_Package_Types
2  public
3  annex EMV2 {**
4  -- error types
5  error types
6          -- errors relative to the Anaogical Switch failures
7          AS_Blocked_Closed : type;
8          AS_Blocked_Open : type;
9          invalid_switch : type;
10         -- errors relative to the Anaogical Switch failures
11         EV_Blocked_Closed : type ;
12         EV_Blocked_Open : type;
13         -- errors relative to the cylinders failures
```

```
14          Cylinder_Blocked_Down: type ;
15          Cylinder_Blocked_High : type ;
16          Cylinder_Blocked_Intermediate : type ;
17          Gear_Blocked_Extended renames type Cylinder_Blocked_Down;
18          Gear_Blocked_Retracted renames type Cylinder_Blocked_High;
19          Gear_Blocked_Intermediate renames type Cylinder_Blocked_Intermediate;
20          Blocked : type;
21          -- system related errors
22          No_Hydraulic_Pressure : type;
23          No_Electrical_Power : type;
24          Anomaly : type;
25  end types;
26  ...
27  **};
28  end Error_Package_Types;
```

Then, we defined the package policy to present all error behaviors related to different equipements such as the analogical switch, the electro-valves and the cylinders correspondant both for closing/opening doors and retracting/extending gears. In the Listing 2, we present a piece of the annex describing the error behavior of the analogical switch device. In this case, we describe different error events. These specified events trigger transitions between different states responsible for describing the analogical switch states and architecture. This can influence the behavior or state of other components that are connected to the analogical switch.

Listing 2: AADL Specification of the Analogical switch error behavior, task

```
1   Annex EMV2{**
2   -- error behaviors
3   -- analogical switch behavior
4   error behavior AS_Behavior
5   use types Error_Package_Types;
6   events
7           Block_open_event: error event;
8           Block_closed_event: error event;
9           Blocked_event       : error event {anomaly};
10  states
11          Open: initial state;
12          Closed: state;
13          Blocked_Open: state;
14          Blocked_Closed: state;
15  transitions
16          open_to_blocked    : Open-[Block_open_event]-> Blocked_Open ;
17          closed_to_blocked : Closed -[Block_closed_event]-> Blocked_Closed ;
18  end behavior;
```

After that, we implemented each of these behaviors in the specific components by adding some instructions to detail error propagation and detections within each component. In case

of transient faults, we are invited to describe recovery statements to follow. Whereas, in case of permanent faults, we have to reinitialize the system, as detailed in the listing 3.

Listing 3: Specification of the analogical switch error behavior, task

```
device implementation analogical_Switch.impl
Annex EMV2{**
use types Error_Package_Types;
use behavior Error_Package_Behaviors::AS_Behavior;
error propagations
        outport : out propagation {No_Electrical_Power, Anomaly};
flows
        esource_1: error source outport{Anomaly} when Blocked_Open;
        esource_2: error source outport{Anomaly} when Blocked_Closed;
end propagations;
component error behavior
propagations
        Blocked_Open -[]-> outport (No_Electrical_Power);
        Blocked_Closed -[]-> outport(Anomaly);
detections
        all -[]->self.Invalid_Switch!;
end component;
properties
        EMV2::Persistence => Permanent;
**};
```

Finally, to describe the behavior of the digital part in case of errors, we used the "AADL Behavioral Annex" to detail in this case the instructions to follow to recover the system in case of sensor error by applying the consensus algorithm to make the right decision from the identical computing modules. In the Listing 4, we present the voter behavior implemented with "AADL Behavioral Annex".

Listing 4: Specification of the Voter behavior, task

```
package Generic_Voter
...
subprogram Voter
features
        x1 : in parameter generic_data;
        x2 : in parameter generic_data;
        x3 : in parameter generic_data;
        output : out parameter generic_data;
        anomaly : requires data access Data_Sets::anomaly;
end Voter;
subprogram implementation Voter.impl
annex behavior_specification {**
variables
        x1_valid : Base_Types::boolean ;
        x2_valid : Base_Types::boolean;
```

```
16          x3_valid : Base_Types::boolean;
17  states
18          all_valid: initial state;
19          x1_invalid : state;
20          x2_invalid : state;
21          x3_invalid : state;
22          anomaly_state : final state;
23  transitions
24          Safety_transition:
25          all_valid-[x1_valid and x2_valid and x3_valid and x2=x1 and x1=x3]->
26          all_valid {output := x1};
27
28          from_allvalid_to_X1_invalid:
29          all_valid -[x1_valid and x2_valid and x3_valid and x2=x3 and not (x1=x3)]->
30          x1_invalid{output := x2; x1_valid := false};
31          from_allvalid_to_X2_invalid:
32          all_valid -[x1_valid and x2_valid and x3_valid and x1=x3 and not (x2=x1)]->
33          x2_invalid {output := x1; x2_valid := false};
34          from_x2_invalid_to_x2_invalid :
35          x2_invalid -[x1_valid and x3_valid and  x1=x3 ]->
36          x2_invalid {output := x1};
37          from_x2_invalid_to_anomaly_2 :
38          x2_invalid -[x1_valid and x3_valid and not (x1=x3) ]-> anomaly_state;
39          ...
40  **};
41  end Voter.impl;
42  end Generic_Voter;
```

As we have already mentioned, we aim at generating middleware components related to consensus, fault detector and fault recovery services to guarantee fault tolerant code generation. In this context, this specification of the consensus will be used in the next step as generic components in the middleware.

6 Conclusion and future work

In this paper, we presented a development process to integrate fault tolerance concerns since the modeling stage. We chose the AADL language to model our system as it gives the users the opportunity to describe functional concerns as well as cross-cutting ones like fault tolerance preoccupations. We decided to model the latter using the Error Model Annex that offers the specification of all classes of faults, its propagation, its detection as well as its recovery. After that, we proposed code generators for both preoccupations. We considered the aspect oriented programming for the implementation of fault tolerance requirements through extending and optimizing the aspect language AspectAda with respect to real-time requirements. We implemented our AspectAda compiler and test it compliance with real time constraints. Then, to validate our approach, we presented an AADL model of a landing system. This model is extended by "AADL Error Model Annex" and "AADL Behavioral

Annex" to respectively model faults, fault propagation, component error behaviors and consensus algorithm applied to make the right decision in case of error.

As future work, our purpose is to complete the definition of requirements and properties needed for building and configuration of our fault tolerant distributed embedded middleware. Then, we aim at accomplishing the development of code generators for both functional and cross-cutting concerns to reduce the cost of the application implementation. Our final goal is to find a trade-off between risk and cost of fault tolerance without degrading system performance.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats - a taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[3] Jingshu Chen and Sandeep Kulkarni. Effectiveness of transition systems to model faults. In *ACM/IEEE the 5th International Conference on Distributed Smart Cameras*, Gent, Belgium, 2011.

[4] Brahim Hamid, Ansgar Radermacher, Patrick Vanuxeem, Agnes Lanusse, and Sebastien Gerard. A fault-tolerance framework for distributed component systems. In *EUROMICRO-SEAA*, 2008.

[5] Brahim Hamid, Ansgar Radermacher, Agnes Lanusse, Christophe Jouvray, Sébastien Gérard, and François Terrier. Designing fault-tolerant component based applications with a model driven approach. In *SEUS*, 2008.

[6] Lasnier Gilles, Robert Thomas, Pautet Laurent, and Kordon Fabrice. Behavioral modular description of fault tolerant distributed systems with aadl behavioral annex. In *10th Annual International Conference on New Technologies of Distributed Systems (NOTERE), 2010*, pages 17 –24, june 2010.

[7] Ana Elena Rugina. *Dependability modeling and evaluation From AADL to stochastic Petri netss*. PhD thesis, 2007.

[8] Ruben Alexandersson and Johan Karlsson. Fault injection-based assessment of aspect-oriented, implementation of fault tolerance. *International Conference on Dependable Systems and Networks*, pages 303–314, 2011.

[9] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *ACM Proceedings of the AOSD workshop on Aspects, components, and patterns for infrastructure software*, New York, USA, 2008.

[10] Kashif Hameed, Rob Williams, and Jim Smith. Aspect oriented software fault tolerance. In *Proceedings of the World Congress on Engineering WCE London, U.K.*, 2009.

[11] Diana Szentivanyi and Simin Nadjm-Tehrani. Aspects for improvement of performance in fault-tolerant software. *Pacific Rim International Symposium on Dependable Computing*, pages 283–291, 2004.

[12] SAE. *Architecture Analysis & Design Language*, April 2011.

[13] SAE. *Architecture Analysis & Design Language Annex E: Error Model Annex*, September 2011.

[14] Myron Hecht, Alexander Lam, and Chris Vogl. A tool set for integrated software and hardware dependability analysis using the architecture analysis and design language (aadl) and error model annex. In *ICECCS*, pages 361–366, 2011.

[15] Thomas Vergnaud, Bechir Zalila, and Jérôme Hugues. Ocarina: a Compiler for the AADL. Technical report, Telecom Paristech - France, 2006.

[16] Tucker S. Taft, Duff A. Robert, Brukardt L. Randall, Ploedereder Erhard, and Leroy Pascal. *Ada 2005 Reference Manual*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[17] Alan Burns, Brian Dobbing, and George Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of Ada-Europe, LNCS*. Springer-Verlag, 1998.

[18] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[19] Gregor Kiczales a1nd John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[20] Knut H. Pedersen and Constantinos Constantinides. Aspectada: aspect oriented programming for ada95. *Ada Lett.*, pages 79–92, 2005.

[21] Wafa Gabsi, Rahma Bouaziz, and Bechir Zalila. Towards an Aspect Oriented Language Compliant with Real Time Constraints. In *22nd IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, WETICE 2013, Third trac on Adaptive and Reconfigurable Service-oriented and Component-based Applications and Architectures - AROSA 2013*, Hammamet, Tunisia, 2013. IEEE Computer Society.

[22] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Ada-Europe*, pages 237–250, 2009.

[23] Frédéric Boniol and Virginie Wiels. Landing gear system. Technical report, ONERA-Toulouse, France, 2013.

[24] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. *INTERNATIONAL ACM SIGADA CONFERENCE, ATLANTA*, 2004.

[25] Frank Singhoff. The cheddar aadl property sets (release 2.0). Technical report, LISYC technical report number singhoff-03-07, Feb 2007.