Internship Report

30th August 2010

Dependability and FDIR model based analysis

Student: Pierre VALADEAU
Space systems become more and more autonomous and complex, which considerably increases the difficulties related to their validation. Powerful and highly automated analysis techniques should help validating such system specifications and design prior to their implementation. Considering the quality of service which is required from a space system, there is a need in implementing onboard mechanisms allowing the autonomous Failure Detection, Isolation and Recovery (FDIR) of the system preventing by this mean the mission from being interrupted or lost. In this context, the objective of the internship is to investigate on the possible model based approaches that can be used in the studied domain. Both faults propagation, and FDIR procedures shall be expressed thanks to a model based approach and associated properties shall be demonstrated.

Modelling requirements

The role of onboard FDIR mechanisms is to “break” the fault and failure propagation within the system before the occurrence of given event. In order to cope with availability objectives and to ensure the coverage of the identified failure modes, mechanisms are organized in a hierarchical way: equipment, function, application and system level. On the level at which the fault is caught depends the isolation capabilities (diagnosis) and the recovery sanctions (restart of the equipment or switch from the failed equipment to its associated cold redundant one).

It appears that the global FDIR strategy is made of re-usable dedicated (for detection, recovery etc.) blocks scattered in the spacecraft. On the interactions consistency between blocks’ behaviours, which may be linked to timing constraints (fault presence confirmation etc.), depend the compliance with the fault tolerance objectives. Furthermore, the spacecraft health status is expressed according to the notion of modes. From the nominal mode, the appearance and recoveries of faults turn the system into successive degraded modes in which the mission continues despite the reduction of the functional or the fault tolerance. The last degraded mode is called the safe mode in which the mission is interrupted.

As both faults/failures propagation and FDIR strategies shall be managed, it appears that properties will be demonstrated on models from which both structural and behavioural requirements can be expressed:

- Structural requirements address the hierarchical composition of the system, the deployment of software on hardware and reusability/independence of used blocks.
- Behavioural requirements address the level of abstraction needed for expressing, with respect to their timing constraint, both FDIR mechanisms and processes leading to failures.

Modelling fault/failures propagation and FDIR mechanisms

Analysing the modelling languages and associated tools, it appeared that no language meets both modelling and validation objectives: a combination of, at least, two languages was needed. As a consequence, AADL was chosen to comply with structural and behavioural requirements and UPPAL for the validation ones. To avoid redundancies in modelling activities, a need, in term of model transformation from AADL to UPPAL has emerged.

AADL is an architecture description language which offers a way to represent a hierarchy of composite elements (systems) which contain software entities (Processes enclosing threads) which can be mapped on physical processing or storage resources (Processors and memories). Hardware and software entities can exchange data or events using ports linked together by connections. The connections can be mapped over the physical notion of buses to allow the information to flow at a given speed. However, AADL does not natively provide an efficient way of representing the dynamic of processes and threads (even systems). This notion is limited to the notions of modes. Anyway, thanks to the language extension points called annexes, third part developers can enrich the native language with components or behaviours. One of these add-ons is called the behaviour annex and allows defining, thanks to states machines, the dynamic of an AADL entity.

UPPAAL is a modelling language and associated tools developed by the Universities of Uppsala and of Aalborg, dedicated to timed automata. It comprises an interactive simulator and a model-checker and therefore provides the capability to model and analyse the temporal behaviour of systems (in discrete time) and demonstrate formal temporal properties. Note that many other model-checking exist but very few address temporal properties, which are very important to characterise FDIR.

Modelling technique and meta-model transformation

A first modelling approach was built in order to validate the AADL to UPPAAL transformation feasibility considering simple faults cases and FDIR mechanisms. A recursive AADL pattern was designed, the recursion level giving the FDIR level (0 for the equipment level, 1 for the function level etc.). Each level was made of three states machines types (models) built thanks to the behaviour annex:

- The error propagation models which describe the way the faults and failures appear and/or propagate at this level
- The decision models (detection and confirmation parts of the FIDR), which are observers of the error propagation models and ensure the detection/confirmation of faults
The action models ((isolation and recovery parts of the FIDR) which are in charge of the reconfiguration procedures. The action models receive orders from the decision models at the same level or from decision models of upper levels.

The global model being an AADL model, it conforms to the AADL meta-model which describes the construction rules (syntax) of all the AADL models. The transformation algorithm was based on this meta-model saying that if there is an algorithm which transforms the AADL meta-model (possibly with some restrictions) into the UPPAAL meta-model then the transformation is valid for all the AADL models. We developed the transformation algorithm thanks to an object oriented language called Kermeta. However, the UPPAAL meta-model didn’t exist, so we built it.

Though this method helps us at demonstrating properties about propagation and FDIR, it suffers from scalability and usability problems. In fact, as there is a lot of cold redundant devices onboard, faults may be present but dormant. The combinatorial of active and dormant faults becomes rapidly unmanageable as far as the model grows. Besides, the model is dedicated to the dependability and FDIR modelling. Faults propagation is explicitly described with events connections. This was a problem of information redundancy: most of the faults and failures propagation can be deduced from a functional model if it exists.

A layered modelling method has, as a consequence, been built. Based on a functional (as opposed to dysfunctional) model, we introduced implicit modelling semantics in order to deduce the fault and failures propagation (for instance a corruption of an output data may propagate to all the consumers of this data). Moreover, by decoupling dependability concepts of faults, errors and failures in independent models, we defined a dependability component pattern which can be independently attached to each functional element and allows a systematic approach for modelling dependability aspects. The dependability/functional binding and the propagation implicit rules avoid the information redundancy problem. Finally, on the merged model, the FDIR mechanisms could be plugged.

State space explosion and validation

The validation based on the UPPAAL model checker became rapidly impossible. The well known state space explosion problem appeared early in the design. Though a complete and robust enough solution haven’t been found, we allowed in the transformation algorithm to extract fault propagation “branches” based on a reduced set of studied input faults. Each branch can be validated independently. However, sometimes, even one propagation chain was too complex to be verified. In such case, we found a method based on the decoupling of FDIR and dependability aspects and on the notion of stable state at a given instant. This method defines how to divide a given propagation chain in parts. The division is valid if it is proved that it can be found an instant such that, in the divided chain part, the only possible evolutions correspond to failure propagation and all the other components of the whole chain are in a single predictable state. Then, for dependability aspects, the fastest propagation time and, for the FDIR, the slowest one are considered. If the maximum amount of time needed for the FDIR to react properly is less than the minimum amount time needed for the fault to propagate then the FDIR strategy is considered consistent.

The final model built includes classical fault and failures propagation (spacecraft attitude determination) and propagation loops when considering the command and control laws linked to the management of the reactions wheels. For these cases, model checking based validation helps us at verifying the consistency of the FDIR strategies and even at tuning them (monitoring cycles period, for instance). Moreover, an interesting result was to confront FDIR strategies in the case of common mode faults. Implementing the onboard communication buses, we figured that when the communication is lost, a lot of FDIR monitoring reacted because their associated data are lost. This means that if the FDIR mechanisms related to the bus doesn’t recover the communication faster enough, other reconfiguration procedure may occur because other mechanisms should think that their related equipment are lost. Finally, environment constraints are also influent for fault propagation, for instance, a solar panel may be failed and the associated fault won’t be seen by the system if the sun is not visible. To model these kinds of constraints we have implemented and validated an extract of the electrical power distribution. In this example, charge and discharge speed of the battery has been considered.

Conclusion and way forwards

At the end, the objectives of the internship have been met: a modelling method allowing a component based design; the expression of temporal constraints, fault propagation and FDIR properties has been built. This modelling method has been evaluated and properties have been demonstrated both for dependability and FDIR aspects. Anyway, a lot of work remains. First we think that the transformation algorithm shall be enhanced with new propagation deduction capabilities. Moreover, we only dealt with simple fault cases and the modelling technique has to be confronted with complex propagation. Moreover, the propagation chain division method isn’t proved to be robust enough to more complex cases. Finally, the question of the AADL usage pertinence is asked. We really found this modelling language adapted to the problematic. Anyway, the AADL usage is clearly not an industrial tendency as SysML seems to be the emergent solution.

From a personal point of view, I studied the behaviour of the satellite FDIR functions in several contexts such as feedback control, embedded networks or power distribution. Additionally, I worked closely to some emerging aspects considering both system design (model based approach) and computer science (meta-model transformation). In the scope of Master in embedded systems I found the internship really interesting and well adapted.
Acknowledgements

I would like first to thank all the people in Astrium who welcomed me and in particular, Didier Munch, department supervisor, who has accepted my candidature. Moreover, some aspects (administrative etc.) of an internship were hidden from my point of view but asked for investments. I am really grateful to people that offered me the chance to work during five months in perfect conditions even if I don’t know who they are (computer installation etc.). For those I know special thanks to Maria-Christina, for your kindness and your help.

Saying “internship” means that there is one “internship supervisor”. Mine was really important for the realization of this project. Jean-Paul, you gave me a lot of your confidence even before the starting of the internship. Then, though you hadn’t time to waste, you guided me for technical solutions, explained me how things are working on spacecrafts and found solutions on some blocking problems I had. So, what to say? Thank you? Is this good enough? But that is for the professional aspects. I also find important to say that discussing about dependability anecdotes or social/cultural events or thinking about how the world goes were not really quantitative moments, but really good and important ones.

Considering, the realizations made during this internship, some people offered me precious help and advice. In particular, I think to Christel Seguin who is incredibly able to focus on what is or can be wrong in my way of thinking to the problematic. Your rigour and the way you have to understand the problems and provide some advice were really useful, thank you. Ana, though we have sometimes divergent opinions when dealing with model checking, I am really grateful for the time you spent with me, for having pointed out advantages and weaknesses in my models and for providing me some directions to follow.

This report will be the end of two years split between work in embedded software and studies. Sometimes it wasn’t so easy to manage both. However, some people at ISAE made going back to school possible and I think I can measure the efforts they made. In that sense, I would like to thank Janette Cardoso and Michel Chauvin. Even if sometimes I was really unbearable, I am also really grateful for their help. I also address warm regards to Bernard Stumpf and Claire Wallez for all they have done for me on the “front line”.

Obviously, I can’t say a word on these past two years without writing a little something concerning the person who shared this experiment with me. So, Beatrice, it’s now difficult to say something that has not been already said as near as the end is. Just know that in the good and in the bad moments, I always thought and measured well what a chance it was to work with you and how many things I have learnt. I sincerely hope (for us), that your future colleagues in your future work will feel as I have felt during these two years.

I think about a lot of people while writing these lines. However most of them won’t read even a line of this report either because they don’t read English (even mine) or because they aren’t interested in or simply they can’t. So, for those for who “embedded” means nothing or nearly and those I think about now, thank you. Thank you for helping me at putting sometimes a different look on my work, my environment, and at keeping in mind what can be called “important things” and how.

I think that I can’t thank everybody, but I hope that persons I forgot may recognize themselves in one of the previous paragraphs. However, there is always an exception. I have mine. So, I would like to address an important and not so well defined thought to someone that will, for sure, understand.
IV. DEPENDABILITY/FDIR MODELLING METHOD
   A. DISCUSSION OF THE CURRENT MODELLING METHOD
   B. DECOUPLING DEPENDABILITY NOTIONS
      1. Modelling fault dynamic
      2. Modelling error dynamic
      3. Modelling failure dynamic
   C. PROPAGATION CHAIN PATTERN
   D. FAULT AND FAILURE COMPATIBILITY
      1. Deduced failures propagation
         a) Software/Software dependencies
         b) Hardware/Software dependencies
         c) Hardware/Hardware dependencies
      2. Fault/Failure classification
         a) Constraints
         b) Used classification
   E. BINDING FUNCTIONAL, DEPENDABILITY AND FDIR MODELS
      1. Step 1: Build dependability patterns
      2. Step 2: Binding models
   F. MODEL TRANSFORMATION
      1. Algorithm chain
      2. Model extraction
      3. Solving fault/failure compatibility
         a) Hardware/Software resolution
         b) Hardware/Hardware resolution
         c) Software/Software resolution
      4. Behaviour extraction and UPPAAL transformation
   V. DEALING WITH STATES SPACE EXPLOSION
# Acronyms

**Dependability and FDIR model based analysis**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>Architecture Analysis &amp; Design Language</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>AOCS</td>
<td>Attitude and Orbit Control System</td>
</tr>
<tr>
<td>BCDR</td>
<td>Battery Charge and Discharge Regulator</td>
</tr>
<tr>
<td>CCD</td>
<td>Charge Coupled Device</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>EADS</td>
<td>European Aeronautic Defence and Space Company</td>
</tr>
<tr>
<td>EAST</td>
<td>Electronics Architecture and Software technology</td>
</tr>
<tr>
<td>FDIR</td>
<td>Fault Detection, Isolation and Recovery</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Modes and effects analysis</td>
</tr>
<tr>
<td>FOG</td>
<td>Fiber Optical Gyroscope</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>LCL</td>
<td>Latch Current Limiter</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MAC</td>
<td>Magneto coupler</td>
</tr>
<tr>
<td>MAG</td>
<td>Magnetometer</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility</td>
</tr>
<tr>
<td>MPPT</td>
<td>Maximum Power Point Regulator</td>
</tr>
<tr>
<td>OBC</td>
<td>Onboard computer</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RW</td>
<td>Reaction Wheel</td>
</tr>
<tr>
<td>SAS</td>
<td>Sun Acquisition sensor</td>
</tr>
<tr>
<td>SST</td>
<td>Standard Star tracker</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst case execution time</td>
</tr>
<tr>
<td>XMI</td>
<td>Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
<tr>
<td>XSD</td>
<td>Extensible Schema Documentation</td>
</tr>
</tbody>
</table>
I. Introduction

A. Foreword

The Specialized Master in Embedded Systems I have followed these last two years is concluded by a five month internship. This internship shall be compliant with the scope of the computer science, embedded networks, energy, automatic and electronic domains studied. Enclosing these domains, the master proposes some transversal and practical notions well known in the industry. This is the case of the dependability and safety analysis.

Coming from computer science and especially from embedded critical software I worked in the scope of maintenance and maintainability software for aeronautical purposes. I followed the courses of this Specialized Master in order to enlarge my culture of embedded systems and to be able to work at system level in a near future. The dependability, safety and security aspects interest me a lot and I chose to focus on them. However, though safety concepts are strongly present in the aeronautical domain, they are not bounded to it. They are also present when dealing with satellites (mainly oriented to reliability) and are emerging in automotive. In fact, they become more and more relevant as far as the technology allowed the design of complex embedded systems which are responsible for critical (safety relevant) functions.

After approaching the safety aspects in the aeronautical domain, I had the chance to study how safety is or will be considered on the automotive side during the student project on which I worked in the scope of the Master. In order to learn how these concepts are managed in the space domain, I proposed my candidature to EADS Astrium Company. This last, was a response to a submitted seducing subject dealing with the analysis of safety and dependability aspects in space domain thanks to model based approaches. My candidature was accepted.

B. Objectives and constraints

Space systems become more and more autonomous and complex, which considerably increases the difficulties related to their validation. Powerful and highly automated analysis techniques should help validating such system specifications and designs prior to their implementation. Considering the quality of service which is required from a space system, there is a need in implementing onboard mechanisms allowing the autonomous Failure Detection, Isolation and Recovery (FDIR) of the system preventing mission interruption or loss.

This first observation shows clearly that there are two main notions which appear when dealing with the specification of a space system with respect to safety and dependability aspects. First, it has to be known how faults propagate within the system in order to determine where the fault monitoring shall be implemented and what kind of monitoring is needed. At a first level, fault monitoring shall be sufficient to diagnose (isolation) a failure of the system. Moreover, it has to be considered how the system reacts in the presence of a failure (recovery). This means that in order to preserve the safety and the dependability of a space system, some actions must be performed autonomously on-board.

According to their nature, these two aspects, propagation and reaction, are strongly coupled. In fact, a fault occurs while the system is in a given state (or behaviour mode) and the faults propagate according to this state. Besides, a recovery action that occurs in reaction to this fault detection may modify the state of the system (for instance, a recovery action may switch off the power supply of an electronic device). As a consequence, the initial fault propagation may be impacted, maybe it changes or maybe new faults (that were hidden previously) appear.

Today, system specifications are produced in a textual format and result mainly from an analysis made by hand. This raises some problems. First, considering the expression format, writers are obviously not lawyers (who are able to write verdicts which are not ambiguous). As a consequence the semantic of a textual specification might be subject to several interpretations which can be the source of implementation errors. Second, as the complexity of the systems grows, manual analyses become unmanageable, abusive hypotheses might be taken or complex cases might be forgotten. This problem is amplified by the system reconfiguration potential that recovery actions provide. Not only the analyst has to concentrate on the faults propagation in all the possible states of the system, but also to consider all the states resulting from recovery actions while the fault is propagated. Moreover, the question of specification validation is raised. In fact with a classical analysis, fault propagation and recovery strategy cannot be proved to be correct, even tested before the implementation phase.

This is where model-based approaches take place. Textual format for specifications will be conserved in the short term (except for fully automated process) but model based approaches offer an interesting solution for solving the complexity problem, help the designer at building a complete and correct FDIR strategy and at formalizing in a non ambiguous manner the behaviour expected from the implementation.
Answering to this problematic, the objective of my internship is to investigate the possible model based approaches that can be used in the studied domain. This means that both faults propagation, and FDIR procedures shall be expressed in the models. In addition, complex systems will result in complex models. As a consequence a modelling method shall be defined. At the end, both propagation properties and FDIR properties shall be, from the model, provable.

This report is articulated over four mains axes. In a first time, considerations about model based approaches are analysed and a status is made in order to select what are the candidates modelling languages. In a second time, we will show a first approach aiming at validating the transformation of FDIR models into provable models. After, the adaptation of the first approach in order to converge to a possible provable modelling technique is shown. Finally, several classical cases of faults propagation and recovery actions are modelled using the proposed method. Modelling activity and proved results are discussed.

C. Fault Detection, Isolation and Recovery

1. Objectives

This first section describes some generalities about what FDIR mechanisms are and what is commonly called “the” FDIR. After some considerations below about the high level objectives, an example shows how it behaves, according to which main principles and where it is implemented within the spacecraft.

From a general point of view, a satellite is not necessarily visible from a ground station. It shall have high constraints of availability or there are simple but repetitive actions to process in order to maintain this availability. These several and variable constraints (depending on the mission profile), introduce the notion of autonomy. The autonomy is the ability of the spacecraft to deliver its service in absence or with limited intervention from operators, in absence or presence of failures. This last case corresponds to fault tolerance. Depending on the mission, at the design level, requirements considering dependability objectives are expressed. The space system shall be kept autonomous according to its dependability objectives and in particular availability. This means that depending on the mission, the autonomy level is variable. For instance a telecommunication satellite shall have a very high level of autonomy (no or few service interruptions) and a probe a lower one except in some particular phases (e.g. orbit insertion).

2. Attitude acquisition example

In order to introduce how the FDIR is working onboard and how it is implemented and where, let’s takes a simple example extracted from the Attitude and Orbit Control System (AOCS), commonly used in agile satellites. This case illustrates how the attitude is controlled among its three axes. The attitude is determined thanks to a full integrated electronic sensor called a star tracker. Generally, a star tracker is a CCD sensor (or CMOS) which captures images from the sky. These images are compared thanks to an FPGA computing unit with a series of images gathered in a catalogue stored onboard in a non volatile memory.

From a general point of view, positioning (position and attitude) a spacecraft can be summarized as a succession of translations from the original position and a succession of rotations. From a mathematical point of view, the position of the spacecraft can be determined from an original position \( (X_0, Y_0, Z_0) \) on which is applied (multiplied) a set of translations matrices \( \begin{bmatrix} X \end{bmatrix} \) and \( \begin{bmatrix} Y \end{bmatrix} \) and \( \begin{bmatrix} Z \end{bmatrix} \). The Attitude is then determined by applying successive rotation matrixes according to an angle belonging to a given rotation axis \( (\alpha_\alpha, \alpha_\omega, \alpha_\kappa) \). Finally, the spacecraft attitude and position can be expressed, from a matrixes point of view, as follow:

\[
\begin{bmatrix}
1 & 0 & 0 & X_0 \\
0 & 1 & 0 & Y_0 \\
0 & 0 & 1 & Z_0
\end{bmatrix}
\times
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\times
\begin{bmatrix}
\cos(\alpha_\kappa) & 0 & \sin(\alpha_\kappa) & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\times
\begin{bmatrix}
\cos(\alpha_\omega) & \sin(\alpha_\omega) & 0 & 0 \\
-\sin(\alpha_\omega) & \cos(\alpha_\omega) & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\times
\begin{bmatrix}
\cos(\alpha_\alpha) & \sin(\alpha_\alpha) & 0 & 0 \\
-\sin(\alpha_\alpha) & \cos(\alpha_\alpha) & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

Anyway, in order to reduce the computation needs associated with such matrixes, an equivalent representation is used: the quaternion. The definition and usage of quaternion are depicted in annex 9.

3. Hardware design overview

At this step, the type of functional data needed to be acquired in order to determine the attitude is known. Now, let’s focusing on how it is implemented considering only the hardware design first. Traditionally, a single redundant computer hosts the software parts related to the realization of functions which provide specific services to the spacecraft (ex: control of the attitude) and contribute to the realization of the mission. The functions are logically decomposed into sub functions in order to depict a structured functional view of the system architecture and to express the dependencies between equipments (sensors and actuators) and the functional layer. For instance, a typical hardware design of a spacecraft platform for controlling the attitude and the orbit (AOCS) can be seen as follow:
The Onboard Computer (OBC) hosts all the software parts related to the computation of the attitude and the orbit. These computations depend on the following inputs:

- The attitude estimation: computed thanks to the measurements made by the star trackers SST-O (the sensor) and SST-E (the electronic of the sensor)
- The attitude: computed thanks to the gyroscopes measurements FOG-0 (the gyroscope) and FOG-E (the electronic of the gyroscope)
- The absolute position on the orbit: computed thanks to the ground stations position acquired by the DORIS equipment.
- The earth magnetic field direction: sensed thanks to the magnetometers (MAG)
- The sun position: sensed by the sun acquisition sensor SAS

With all these inputs the attitude and the orbit are then applied and maintained thanks to the reaction wheels (RW-M and RW-E), the propulsion system (PROPU) and the magneto couplers (MAC)

4. Functional design overview

From a functional point of view the previous architecture can be decomposed according to the following functions and sub-functions. This decomposition is made according to the notion of “level”. There are four main levels:

- The equipment level
- The equipment functional level
- The application functional level
- The system level

Figure 1: AOCS hardware architecture

Figure 2: AOCS functional decomposition
5. Fault propagation

The illustrative case chosen for introducing to FDIR mechanisms is enclosed inside the hardware/functional architecture depicted briefly before. For this approach, the propagation of faults occurring in a star tracker through the functional design will be studied. First, the theoretical fault propagation will be shown, considering that no FDIR mechanism is implemented and then according to several failure modes that can be reached, the detection, isolation recovery procedures will be highlighted.

Basically the system is made of three star trackers. Two are used by the “estimation” sub-function in order to compute an estimation of the attitude (in the form of a quaternion) of the spacecraft and one is switched off (cold redundancy). As long as the “estimation” sub-function is able to compute the attitude from two star trackers, the AOCS management function behaviour is not degraded. Anyway, it is possible to compute an estimation of the attitude based on a single star tracker, but the behaviour is degraded. Without any star tracker and considering that the attitude cannot be computed by another way, the mission is lost.

For the purpose of this example, the problem is simplified by introducing the notion of threshold. We will simply consider that beyond the threshold, the system is said to have failed. Let’s consider the case that at a given instant the first star tracker starts computing a wrong quaternion. In absence of any FDIR mechanisms, this failure will propagate to the “estimation” sub-function in which the computation of the attitude will start diverging. After a certain amount of time of failure presence, the attitude computation will become completely incorrect (threshold reached) and this failure will propagate to the AOCS management function. The failure will affect the computation of the attitude that drives the actuators and after another time of divergence, if the failure persists, the effective attitude of the spacecraft will become wrong. The propagation of the initial failure is illustrated on the following figure:

![Theoretical fault propagation diagram](image)

**Figure 3**: Theoretical fault propagation

6. FDIR strategies

   a) Fault detection at equipment level

Let’s consider now that the fault which appears is a simple, permanent and detectable fault and that it can be detected at the lowest level of the architecture: at equipment level. The strategy chosen for this case has been built in order to be representative for commonly used techniques for space systems. The FDIR strategy is the following:

- At equipment level the failure is detected and the appearance event is transmitted to upper layer
• At equipment function level: the notification is acquired and the failure is confirmed in two steps:
  o A temporisation is performed
  o If the failure persists after the temporisation the sensor is restarted (reset signal)
  o If the failure persists again a confirmed appearance event is sent to the upper layer
• At application level: the confirmed event is acquired and the reconfiguration occurs:
  o If there is a cold redundant sensor:
    ▪ The failed sensor is switched off
    ▪ The redundant one is switched on
  o If there is still a sensor working:
    ▪ The failed sensor is switched off
  o If the failed sensor is the last one:
    ▪ The failed sensor is switched off

The case of the first failure of such a scenario is illustrated below:

Figure 4 : Full FDIR recovery chain
b) Fault detection at equipment functional level

Now, let’s consider a fault which is not detectable at equipment level but detectable at equipment functional level. In such a case the “estimation” sub-function might detect that one of the two used sensors doesn’t work but cannot determine the failed one. Necessarily, after a confirmation time, a detection event will be sent to the AOCS level which has no other choice but to switch off the two sensors and to start the third one. This behaviour is illustrated below:

Figure 5: detection at equipment functional level

The final fault type concerned is a fault which is neither detectable at equipment level nor at estimation level. In such a case the AOCS management level may behave with an inconsistent attitude estimation and by this means apply wrong set points to command laws (to the reaction wheels for instance) and the spacecraft will enter a non safe state. But there is a high level monitoring made through the sun acquisition sensor. This sensor defines whether or not the sun is well pointed. This kind of measurement is very imprecise. Anyway, if the attitude diverges a lot, the sun acquisition sensor will detect that the sun is no longer pointed. This will be detected at AOCS management level but this last cannot isolate the failed equipment (star trackers,
reaction wheels, or sun acquisition sensor). In such a case the mission shall be interrupted (the satellite enters then in a survival mode and waits for ground intervention).

7. Conclusions on FDIR properties

a) Hierarchy

The first observation when looking at this simple and idealistic example is that the FDIR mechanisms are organized hierarchically in order to ensure the coverage of fault detection (even for those which are not detectable at a low level). Obviously, from the level of detection depends the ability of isolation, but also, from the level at which the fault is managed depends the level of sanction. The level of sanction is defined by the sequence of reconfiguration actions that leads to recover fully or partially the system. And the level of recovery may lead to an interruption of the mission.

From all these observations, the role of such layered FDIR mechanisms is to “break” the error propagation chain before a given event occurs. For instance, if the strategy’s’ objective is to prevent, in case of a single failure, the estimation sub-function from computing a wrong attitude, this means that the reconfiguration process has to be performed in a delay which is less than the time needed after the occurrence of the fault for the estimation to become wrong. By this mean, one of the problematic for designing FDIR is to keep the strategy consistent with the manner that the system propagates faults and failures.

Moreover withdrawing systematically an equipment in case of failure is not a pertinent solution because, the life time of the system is reduced. The environment in which the satellite operates is very aggressive (radiations...). As a consequence, faults are not compulsory permanent but may be transient. In other words, a given device can be altered by a sporadic event and a simple reset of the equipment might be able to solve the problem and restore a nominal behaviour. By this mean, a pertinent detection shall be chosen to allow performing localized actions (reset the device for instance) and to allow not withdrawing abusively the equipment.

b) Modes

As it has been already said, recovering the system changes its state, some devices are switched off, and some other are switched on for instance. But, considering permanent failures, when a device failed, it is no more useable. This means that, for instance, a redundant one is activated. Even though the mission is not perturbed (the service is still delivered), the fault tolerance ability is reduced. This means that, more than the common functional modes (initialization, nominal etc...), there are also dysfunctional modes. Roughly, three categories can be highlighted:

• The nominal modes in which the mission is performed despite the fault tolerance ability might be reduced.
• A set of degraded modes in which the mission is altered and the fault tolerance ability is reduced (according to the severity of the mode).
• A safe mode where the mission cannot be performed. In such a case, the spacecraft keeps alive a minimum set of devices which allow to survive until a ground station analyses and solves the problem manually by sending commands (if possible).

Moreover, though a dysfunctional mode can be applicable to the whole spacecraft (it is the case of the safe mode), it is better to think in terms of combination of modes. For instance, considering that the star tracker sends the measurement to the...
estimation sub-function thanks to a cold redundant bus. If there is a failure on the bus, the communication is in degraded mode (only one bus remains) but the communication is still efficient so at AOCS level, the nominal mode is preserved. For the example seen before, the combination of modes can be summarized as follow:

<table>
<thead>
<tr>
<th>Device</th>
<th>State</th>
<th>Onboard communication</th>
<th>Estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Nominal</td>
<td>Deg</td>
</tr>
<tr>
<td>SST</td>
<td>All SST working</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SST</td>
<td>One lost SST</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SST</td>
<td>Two lost SST</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SST</td>
<td>Thee lost SST</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Bus</td>
<td>All buses working</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Bus</td>
<td>One lost bus</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Bus</td>
<td>Two lost bus</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**c) Spreading**

From the example, it can also be seen, that the FDIR cannot be considered as single software in charge of maintaining the autonomy of the spacecraft. Though there is a centralized FDIR software, FDIR mechanisms (hardware and software) are also spread in several locations, all the parts contributing at preserving the availability of the whole system. Moreover, FDIR mechanisms are mixed sometimes with what can be called the functional design in an ambiguous way. For instance, let’s take a simple C function:

```c
void f() {
    unsigned int _acquisition_result = 0;
    t_quaternion _quat;
    _acquisition_result = acquire_attitude(&_quat);
    if (_acquisition_result == (int)ACQUISITION_OK) {
        /* Program branch dedicated to functional treatments
           * => nominal actions are performed*/
    } else { /* Program branch dedicated to FDIR treatments
               * => an incorrect acquisition has been detected*/
    }
}
```

This function could be a member of the estimation software part, but as it can be seen, a detection mechanism is implemented. Despite the fact that this software part is a pure functional one, it also includes a certain level of detection and of subsequent actions which is a part of the FDIR domain. In any case, this notion shall be kept in mind while modelling: the frontier between functional and FDIR notions aren’t, at implementation level, simply decoupled.

Moreover, some FDIR actions which have a direct impact on the mission are not compulsory taken at high levels of the functional chains. In most of such cases, they can’t because they operate in order to preserve the integrity of the satellite. For instance, in case of an over current detection at the device level, the fuse will burn. This action may have an impact on the mission, because the device is no more useable. Anyway, this protection is very important because an over current consumption may damage other devices: the term of “reflex” action is often used.

**d) Time constraints**

The last important notion observed is the notion of time. As it has been shown, the propagation is timed based. Though in a static analysis the propagation can be considered instantaneous by studying the potential impact all along the propagation chain, it becomes not possible when dealing with recovery. In fact, after the detection, the confirmation may take time, and from this amount of time depends the moment at which the reconfiguration occurs. Moreover, the recovery action may also take a given amount of time. In addition, FDIR components are not necessarily linked with communication events. In fact, they could also be synchronized by delays (the lowest level having the smallest delay). All these temporal aspects shall impact the fault tolerance ability and event fault propagation itself.

**D. Conclusions on model properties**

At the end, of the previous observations, some directions are clearly highlighted concerning what the model based approach about to be designed shall allow. These observations drove the choices which have been made during the internship. First, the example studied loops back with the representation generally made when introducing the FDIR concept. The scheme below illustrates a summary of what has been said in more a generic way.
1. Structural requirements

a) Deployment requirement

When thinking about fault propagation, it appears clearly that they occur according to two main axes. Though, it has not been depicted in the previous sections, the first one will be called the vertical one. In fact, looking at how the star tracker may fail, it has to be considered that one possible fault propagation chain is that a fault occurred in the FPGA is propagated to the software acquisition routine which leads the measurement to be erroneous. This kind of propagation implies the introduction of the deployment notion. In fact, the model shall allow identifying clearly on which hardware solution is hosted software solutions. Moreover, this notion is not limited to the hardware software deployment. From a general point of view It is applicable when the level of abstraction changes. For instance, a failure in a sub-routine of a software element leads to the failure of the whole software. In term of functional decomposition, there is also a vertical propagation.

Moreover, when the fault propagation reaches the software parts of the design, faults shall be propagated within the several software parts which contribute to deliver the service concerned. This kind of propagation is said to be horizontal in the sense that it concerns the functional dependencies.

b) Compositional requirement

One of the goal of the study is to help at building simple models (i.e. complex model shall be easily built). In order to reach this objective the idea of creating predefined and extensible modelling blocks which can be composed seems simple. The notion is close to the notion of component, in term of modelling technique. In fact, it is expected from modelling entities to be reusable in several contexts. Ideally, components libraries shall be built and designers will only have to plug existing components in their system models. As a consequence, the notion of component has to be defined in order to answers to tow main questions:

- How a component is structured (internal design and composition interfaces)?
- What are the composition rules between components in order to ensure the consistency of the whole model?

2. Behavioural requirements

a) Abstraction

From a dependability/FDIR point of view, at design level, it looks obvious, that the behaviour of components shall be defined. In other words, fault propagation logics, detection and recovery actions shall be parts of the model. However, a precise and complete behaviour is not expected. For instance, it is not necessary to know exactly why the attitude diverges in five seconds in the presence of a fault. The important notion considering dependability is that after five seconds the attitude has diverged. The modelling solution must provide a way to express behaviour at the correct level of abstraction.

b) Time expression

This is maybe the most important requirement. It doesn’t concerns directly the modelling technique but it as an impact which cannot be neglected on the solution chosen for validation purposes. In fact, as it has been shown, the notion of time will be
recurrent during all the modelling activity, this means that both the modelling language and the validation solution must consider and be compliant with this dimension.

II. Model based approaches

The goal of this section is to make a quick overview of the model based approaches. This analysis will drive the selection of the models used for the study as well as of the tools needed to reach the objectives. In a first time, the expectations about model based approaches are defined. Then, a quick overview of modelling techniques is done and finally the selection of modelling techniques and tools used and developed is justified.

A. Modelling activity

1. Modelling dependability/FDIR

The model based approach notion dealing with dependability is sometimes ambiguous. If a model is a manner of representing the system according to a given point of view and abstracting away unnecessary details, the, dependability process has always dealt with models. For instance, a Failure Modes and effect Analysis (FMEA) table is a model: It is a structured representation of the system focussing on the effects of failures. The questions which must be asked are:

- Can such a model suffer from different interpretations?
- Is the model representative for the system being designed?
- Is such a model exploitable for automatic demonstration?
- Is the model simple to be built?

In the scope of the study we will try to use modelling techniques and/or languages which may positively answer to these two questions.

a) Modelling objectives

Dependability and FDIR analysis/design requires different categories of support depending on the analysis to be made. For each kind of objectives modelling methods exist. This section will recall briefly the different types of analysis which may require model based approaches for correctness, completeness or efficiency reasons.

- Quantitative (probabilistic) assessment of dependability properties: The purpose is to evaluate global probabilistic properties of a space system (or a part of it) based on its architecture (especially in terms of redundancies) and a set of assumptions on the stochastic distribution of failures that can affect the system and its elements. The relevance of such an analysis is anyway mainly linked to the relevance of the input failure rates rather than on the ability to combine them to highlight a given property at system level.
- Qualitative (descriptive and/or deterministic) assessment of propagation of faults and failures and of the effects of this propagation: This corresponds to a very important and large set of analyses of the effectiveness of detection and protection mechanisms against faults and their combinations, analyses of common mode failures or common cause faults and their effects, etc. This is one of the major targets of the study considering the importance of the associated objectives and the limitations of the currently used techniques
- Assessment (correctness, performance) of FDIR: For correctness and/or performance assessment of any function, model-based approaches can be used to express the behaviour according to the resources needed by the function. Such modelling technique can check the compliance of the function with a given target of evaluation. For instance, memory consumption or worst case execution time (WCET) could be determined as well as pure behavioural response in the presence of specific inputs.

b) Scope of the study

The study we made doesn’t focus on the quantitative aspects. In fact according to the high level objectives seen before, the modelling activity will focus on the fault propagation, and the recovery aspects. This means that, the study will address the qualitative and correctness categories. However, dedicated models are not necessarily the best solution. It could be interesting to exploit the information coming from engineering models (functional system design).

Considering this scope and what has been already said concerning the model requirements, it appears that two mains categories of models shall be used:

- Structural models which are often used as a support to the engineering activities, to represent the organisation of the various elements composing a system (at various abstract levels: functions, sub-systems, equipment, software, etc.).
Structural models can be extended to represent how the faults affect elements of the structure, and how these faults propagate along the structure.

- Behavioural models which are often used as a support to the engineering activities. In particular they are used because they come with associated tools allowing formal verification of behavioural properties. Such models are generally based on automata (or inherited).

2. Modelling languages

According to the purpose of the study drove by what is required for FDIR modelling and what are the objectives of the modelling activity, modelling technique(s) has (ve) to be chosen. This section summarizes some model based approaches studied in order to select the most relevant as far as FDIR is concerned.

a) UML/SysML

UML (Unified Modelling Language) is very popular in software engineering, standardised by the Object Management Group (OMG). It provides several “views” (structure, state machines, classes, etc.) in order to express the whole design relevant phase of software engineering. UML is not only a model-based approach in the sense of it provides model syntaxes and semantics; it is also a development process. In fact, designers shall describe what are the services expected from the system and what are the connections with the different users in a use cases diagram. Then, the use cases have to be refined to build what is called the static conception (class diagram) where the structural part of the design is finalized. Then, the designers shall focus on the dynamic parts. For describing how the system behaves, a set of diagrams are available each being appropriate for specific purposes: activity diagram for high level or multi-processing description and sequence diagram (close to Lamport diagram) for protocols. The last part of the design concerns how the software is physically built: what are the libraries, the data bases, and the executables are described in a components diagram. Finally the last part concerns how the software is deployed (on which machine a part of the component diagram is deployed). This is made through a deployment diagram.

SysML appears as both a specialisation and an extension of UML SysML supports a more formal semantics, in particular adapted to system and real-time modelling. Based on similar concepts and presented as a profile of UML, it is supported by several UML tools. It is potentially an interesting candidate for the study because of its suitability for system and software engineering activities and its capability to represent the notions of interest for the safety and dependability analyses. In fact, it allows the designer not to be concerned by pure software consideration. That’s why, for instance, the term “class” disappeared to be replaced by the notion of “block”. SysML introduces also some hybridization notion such as continuous variables.

For both approaches many tools are available. Anyway as far as formal temporal verification is concerned, UML/SysML modelling tools only provide structural verification tools (based on Object Constraint Language (OCL)) or simulation tools (diagrams animation). We didn’t manage to find relevant tools highlighting behaviour evidences.

b) EAST ADL

EAST-ADL is a domain-specific (automotive) architecture description language, and aligned with the AUTOSAR execution framework. As an ADL (architecture description language), it provides a compulsory manner of representing what is an item (process, thread, device, processor etc.) of the system. Roughly, an item is described by one and only one interface which describes the interactions possible with other components. Then, the interfaces could be implemented several times. An implementation is in fact the description of what is contained inside the item (subcomponents). Based on these three notions (interface, implementation, and subcomponent) the model can be composed by using items implementations as part of other items subcomponents.

Anyway, EAST ADL is really related to the automotive domain and, as a consequence, aims at describing the whole design phase: from commercial issues to the configuration table loaded onboard on AUTOSAR compliant computers. In that sense, it proposes a layered model which can be seen as follows:

![Figure 8: EAST ADL development process](image-url)
c) **AltaRica**

AltaRica is a language developed by the LaBRI laboratory (Bordeaux, France) to model both functional and dependability aspects of systems. Some academic and industrial tools exist to design and analyse AltaRica models, including simulators and model-checkers. In particular Dassault (initially Dassault Aviation, then Dassault Systems) and EADS APSYS developed such tools and used them for dependability and safety assessment of critical complex systems.

From a general point of view, an AltaRica model (i.e. a way to structure parts of the AltaRica language) is a transition based model. This means that a component is described by an interface and a transitions system. The evolution of information carried by the input connections may fire transitions and modify the information carried by output connections. Moreover, synchronization based communication and events priorities are also implemented to complete the component based approach. AltaRica is really useful to study faults propagation in large scaled systems such as aircrafts because the associated tools provide on the one hand a way to generate sequences and cut sets which are relevant for safety targets and in the other hand a model checker to prove that safety requirements are met. Anyway, except some isolated studies, AltaRica doesn't allow to model the notion of time.

d) **UPPAAL**

UPPAAL is a modelling language and associated tools developed by the Universities of Uppsala and of Aalborg, dedicated to timed automata. It comprises an interactive simulator and a model-checker and therefore provides the capability to model and analyse the temporal behaviour of systems (in discrete time) and demonstrate formal temporal properties. It is provided under various licensing schemes (from commercial to freeware according to the kind of utilisation and user). Despite its academic nature, it seems to be currently one of the best, if not the only reasonably mature, tool for timed automata. Note that many other model-checking tools exist but very few address temporal properties, which are very important to characterise FDIR (at least properties on the sequencing of events, if not really on temporal behaviour in the sense of the physical time).

Anyway, native UPPAAL doesn’t express a structured hierarchy of the system. This means that the model describes at once the functional properties of the system inside its structural environment. In other words, there is no simple way to decouple the behaviour and the structure. However, synchronizations are possible by splitting the model into many timed automata. In such cases, communication is ensured by the notion of channels, channels having strong semantic properties:

- Synchronizations occur only if both sender and receivers are ready to synchronize (no FIFO or buffer on the receivers side)
- Synchronizations occur without any constraint on the receivers if the associated channel is said to be broadcasted.

e) **SCADE**

Control engineering specific approaches and languages exist to support the modelling, analysis and development of systems having to interact with the physical world and take into account physical phenomena (not necessarily modelled as continuous phenomena when a discrete approximation is sufficient). Scade is a development environment based on the Lustre synchronous programming language. Developed by Esterel Technologies with a certified code generator, it is used in particular by Airbus. Some recent studies have addressed the capability to extend Scade models so as to express dependability related attributes and perform dependability analyses.

Control engineering approaches are in principle interesting, first because of their use in system and software engineering and second because many phenomena of interest for dependability are real world physical phenomena for which dedicated detailed models are useful. However it is generally sufficient to use such models for dedicated studies of some particular phenomena without trying to interconnect these models to other complex models of the structure and behaviour of the complete system.

f) **AADL**

AADL is a very rich architecture description language in which can be expressed both the structural and the behavioural aspects of a system. Roughly it offers a way to represent a hierarchy of composite elements (systems) which contain software entities (processes and threads) which can be mapped on physical processing resources (processors and memories). Hardware and software entities can exchange data or events using ports linked together by connections. The connections can be mapped over the physical notion of buses to allow the information to flow at a given frames rate. AADL addresses hardware, software and composite items of the system. The AADL elements and their associated syntax are defined in the annex 10.

However, AADL does not natively offer an efficient way to represent the dynamic of processes and threads (even systems). This notion is limited to the notions of modes. The switching between modes occurs on the reception of events. Anyway, AADL provides the notion of annexes which are in fact a point of extension for the language. Third party developers can add extensions to the language. One of these add-ons is called the behaviour annex and allows defining, thanks to states machines, the dynamic of an AADL entity.
In addition, AADL provides the notion of properties. A property is used to enrich an AADL element (or a list of AADL elements). The property values can be native types (Boolean, Integer), enumeration, or references to another component in the system. Natively AADL provides a set of predefined properties, but when modelling, the designer may create custom ones. As an example, the software on hardware deployment is implemented thanks to properties. The predefined property “Actual_Processor_Binding” below, illustrates how the process “process1” is attached to the processor “proc1”:

```
Actual_Processor_Binding => reference proc1 applies to process1;
```

AADL modelling tools come with a lot of analysis facilities: Completeness of the model, flow latency, resource consumption, schedulability can be analysed. Anyway, no AADL tool provides behaviour model checkers.

**g) Summary**

The figure below illustrates the first conclusions which can be made concerning the several model based approach seen.

<table>
<thead>
<tr>
<th>Language</th>
<th>Models</th>
<th>Requirements</th>
<th>Evidence highlights</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AltaRica</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SysML/UML</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Scade</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPPAAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 9: languages summary](image)

**B. Chosen solution**

1. **Modelling languages selected**

After analyzing several modelling techniques, it appears that no candidate is able to fulfil all the requirements related to the fault propagation and FDIR modelling and validation. In fact, model-based approaches are either good for modelling (both structural and behavioural domains) or good for proving but rarely both. AltaRica and Scade can be possible candidates. But AltaRica doesn’t model time and we think that replacing time expressions by transitions priorities may be too complex. Scade solves this problem and according to our opinion could be a really good model to express local fault propagation and isolated FDIR mechanisms, but the deployment notion is missing and it has not to be forgotten that Scade was build in order to support control engineering (even certifiable generated code). This raises the problem of abstraction because what we need is the ability to express dependability and FDIR properties at system level (high level).

Concluding on previous points, we have to think about a combination of models in order to meet the objectives. However, it is not a point, if we think about a potential use of the study, to oblige the system designers to face with two modelling techniques in order to be able to validate a FDIR strategy for instance; building two models to express the same things is not acceptable. Anyway, the solution appears clearly. Concerning validation, UPPAAL seems to be the only accessible tools we have to reach the validation objective because it comes with formal temporal verification tools. So, we have to select a modelling language that is compliant with all the modelling needs except the validation one. Then, we will have to transform this first model into the equivalent UPPAAL one in order to perform model checking. The model transformation need appears now and will be a key point for the study.

Now we have to select a modelling language which will be in front of the system designer. Two solutions remain: AADL and UML/SysML. The two solutions are very close from each other but we chose to select AADL for a practical reason. In fact, SysML is an abstract language (inherited from UML) and it may be difficult to be appreciated by future users. In the contrary AADL offers an interesting level of abstraction, it deals with systems, processors, memories, processes, communication flows which are familiar notions close to the real design of the system. Moreover AADL provides natively a compositional approach, the connections between components being described by their interfaces whatever their content. In addition, we may have to deal with engineering model in order to analyse the fault and failure propagation. This means that the model will be close to the architecture of the system. In terms of simplicity it looks better to use an architecture dedicated language (AADL) rather than a more general system modelling one.

At the end, the study will be oriented around two main languages AADL and UPPAAL. Anyway the problem of transforming one into the other is asked. Concerning the tools used a complete installation guide can be found in annex 1. We have now to study the existing contributions which have been made in this frame (AADL, UPPAAL and transformation)
2. Related works

a) Around AADL

(1) Error model annex

Concerning dependability aspects, several contributions have been done around AADL. First, we have already introduced the reader to the notion of AADL extension points: the annexes. One of them is called the error model annex and is depicted clearly in the document referenced in Bibliography [2]. The idea of this annex is to provide temporal and stochastic models which express the way a given component fails. Such a model is called an error model and encloses the notion of faults appearance (according to their probability), the process needed for the faults to be propagated within the component and the way the failures are propagated to other components. This kind of model takes the form of automata which an example is shown below (this figure is an extract from Bibliography [2]).

![Error model annex automaton](image)

Moreover, faults and failures are propagated according to dedicated flow ports called “error propagation”. This kind of ports introduces a strong semantic because error models are not explicitly connected thanks to these ports. In facts, the error model annex introduces the notion of implicit propagation between models. A set of rules has been defined to express how faults and failures are propagated. For instance, a failure is propagated to all the components linked to the root one thanks to its outgoing connections.

Moreover, the error models can be enriched with guards. A guard is a conditional expression which could stop or allow a failure to be propagated. A guard can be placed at the input of a component (to prevent the fault from altering the component) or at the output of the component (defining that the failure is confined inside the component).

However, the error model annex doesn’t come with exploitation tools (or accessible). Indeed, some additional contributions show that it is possible to transform error models into stochastic Petri nets on which simulation and validation can be done (cf. Bibliography [3]). Anyway, the error model annex cannot be used locally in an AADL model, the whole concept shall be considered and in particular when dealing with the exploitation of the model all the implicit propagation rules have to be considered. If not, the transformed model will not be equivalent (in terms of propagation) to the original one. In addition, scalability problem relative to the usage of Petri Nets seems to be difficult to solve.

(2) Behaviour annex

When looking at model based approaches benched in space domain, we note that contributions around AADL and especially around the behaviour annex have been done (cf. Bibliography [4]). Relative to the expression of rigorous specification of the onboard software and especially the AOCS, the potential of the behaviour annex has been shown. Mainly, we found that behaviour annex expresses more state machines rather than automaton, in the sense that the “states” of the state machines can be composite states (can be described by another automaton). We note that the state machines which can be defined are of the form:

```plaintext
states
   s0 : initial state;
   s1, s2, s3 : state;
transitions
   s0 −{ timer ? }> s1;
   s1 −{ init ? }> s2 { ack ! ; };
```
s2 −{ init ? }−> s3 { nack ! ; }

s3 −{ eof ? }−> s0;

This simple example shows how to define states and transitions. Anyway, it is possible to define composite states. A composite state allows defining a sub automaton which is only relative to this composite state. Moreover, it is shown that the behaviour annex can control the firing of events which are members on the component interface being defined. In the previous example, “timer”, “init” and “eof” are input events waited thanks to the lexical item ‘?’ and “ack” and “nack” are output events fired thanks to the lexical item ‘!’.

What is also important to point out is the nature of the transition. Transitions are of the form: State_initial −{ guard }−>State_final { action }; where the guard can be linked to an event or can express a temporal aspect (for instance [on timeout 500 ms]). The action field has the same properties. In the action field can be expressed, for instance, delays (delay (10 ms, 20 ms)) or computation times (computation (10 ms, 20 ms)). Thinking about the transformation to UPPAAL models these notion will have to be transformed into the notion of guard, synchronization and invariant (for states).

(3) AADL to AltaRica

The question of how to exploit an AADL model for safety and dependability purposes has already been raised. Some contributions go in that direction in order to use the AltaRica language to take advantages of its associated tools such as fault trees or model checking ability. The study presented in Bibliography [5] shows how AADL behaviour annex state machines are transformed into AltaRica transition systems. However, the transformation shown seems to be uncompleteness or limited in the sense of it doesn’t deal with temporal expression which can be part of the behaviour annex state machines.

b) Around UPPAAL

On the validation side several studies considered the validation potential of UPPAAL. In safety and dependability in space domain two main studies have been done (Cf. Bibliography [6]). The first one deals with the validation of safety objectives addressed to the FDIR. The FDIR strategy is based on generic spacecraft demonstrator architecture called AGATA. This architecture must allow implementing the autonomous satellite architecture and simulating the behaviour of the global system (including FDIR) in order to validate on ground the design and operation of such systems. Moreover, the model checking based validation has been made on a significant part of the system but not on the whole architecture. In fact UPPAAL model checker is limited as far as the model size is concerned. The notion of size for the UPPAAL model is difficult to define because the resolution time of the properties is not linear with respect to a simple criterion. It depends on the property to check and on the model. Anyway, with the extract chosen safety targets had been verified:

- The failure of a device that is necessary in a given nominal mode leads the function in the appropriate degraded mode immediately.
- A single failure does not lead to Safe mode.
- Safe mode of sub function implies Safe mode of function.
- Occurrence of several failures leads to Safe mode.
- In Safe mode, only the equipment necessary in Safe mode are switched on.

The state space explosion appears clearly in the second case studied which is the case of formation flying validation. In this context, the objective was to validate the entire control command strategy for spacecrafts which are flying close from each other. In such a case the minimal model relative to the problem is quite huge (49 automata ranging from 2 to 10 states) and is impossible to be verified by the model checker. At the end, the validation has been made in two steps:

- A simulation has been performed on the whole model
- The model checking based validation has been made by extracting, by hand, several sub-models which have been validated independently.

In any case, the problem of state space explosion will, for sure, be a problem in our study. In fact, if we roughly fix the computation limit to 50 automata, dealing with complex models will make this limit being exceeded. This has to be kept in mind.

C. Model transformation

As it has been said, if we want our modelling strategy to be useful, the dependability/FDIR engineers shall have only to deal with one modelling tool. This raises the question of model transformation. Considering that AADL will be the user interface and UPPAAL will be only used for validation purpose the transformation which has an interest is the transformation from AADL to UPPAAL, the reverse operation won’t be useful. In fact, the user won’t have to modify the generated UPPAAL model. This has a major consequence. In fact, we won’t be obliged to preserve all the properties of the initial model after the transformation in order to be able to process backward. Obviously, all the behaviour shall be preserved as well as all the interactions. Furthermore, the generated model shall be understandable enough to be used for simulation purposes. Anyway, AADL allows to build a
hierarchy of elements, UPPAAL doesn’t. This hierarchy can be lost without any impact (with respect to the conservation of all the interactions). This section will be dedicated to the introduction of the relevant notions, tools and language which are involved in model transformation.

1. **Meta-modelling**

In order to be more precise, AADL is a model based language. This means that, the syntax of this language is specified using a meta-model. A meta-model is a model written in a simple language (close to UML) called MOF (Meta-Object Facility). The object oriented concepts are limited in this language and are bound to the notions of inheritances and associations (including aggregation and composition). The goal of a meta-model is to describe formally the rules of conformance for the construction of a model. The notion of conformance is the key one. A model is said to be conform to its associated meta-model. The relation between model and meta-model is shown below:

<table>
<thead>
<tr>
<th>The Graph G1 conforms to the metamodel MGraph</th>
<th>The metamodel MGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Graph Diagram" /></td>
<td><img src="image" alt="Meta-model Diagram" /></td>
</tr>
</tbody>
</table>

**Figure 11 : meta-model conformance**

The simple example above shows that to conform to the metamodel MGraph a model shall be made of two sets of ENode and EEdge:

- Each node shall have a single string representing its name
- Each node shall have a single Boolean indicating if the node is an initial node
- Each edge shall have a single string representing its name
- Each edge shall reference one and only one node as its source (ESource)
- Each edge shall reference one and only one node as its target (ETarget)

The graph G1 conforms to these rules:

- a set of named nodes : { idle, start, run } where idle is initial and the others are not initial
- a set of named edges: {start cmd, run cmd, stop cmd}
- each edge as single source and a single target

Indeed, the question of verification tools to validate the conformance of a given model G, to the meta-model MGraph is raised. The OMG (Object Management Group) defines a standardized method based on the mark-up language XMI to reach this objective. In fact a model can be written in the XMI language. The XMI language allows referencing a meta-model that verifies the construction. Then an XMI parser is able to say if the model is well formed. This principle is nearly the same as what could be found using XML and XML style sheets (XSD).

Dealing with AADL, the principle is the same as the previous small example. The AADL models are written in XMI and the construction rules during design are checked thanks to the whole AADL meta-model. What is important to point out is a consequence concerning the rules which could be given to the final user. For instance, maybe we will focus only on one part of the AADL language or maybe we will not tolerate in transformation all the construction rules allowed by the language. As a consequence, if we want to restrict the usage of AADL to a set of simpler construction rules, the only thing we have to do is to delete from the AADL meta-model useless construction types. The conformance of the constructions will be checked with the simplified version of the meta-model, and any forbidden design (which uses more AADL tips than allowed) will lead to a non conformance.
2. Model transformation

At this step we have introduced the notion of conformance of a model to its meta-model. Based on such a consideration, a robust transformation is possible. In fact, if we are able to develop a complete transformation which can turn an AADL meta-model into an UPPAAL meta-model then all the AADL models (conforming to the AADL meta-model) can be transformed into an UPPAAL model (conforming to the UPPAAL meta-model).

Several tools for parsing and writing meta-models are available; we chose to use Kermeta (provided by an eclipse plug-in) because the transformation language provided is close to Java (simpler than ATL rules) and because it is possible to manage models described by several meta models (it is the case with the AADL models which are described both by the structural AADL meta-models and by the behaviour annex meta-model). The transformation process is illustrated below and the complete Kermeta reference manual is referenced in the bibliography [7].

3. UPPAAL Meta model

Unfortunately we could not find any meta-model available for UPPAAL configurations. As a consequence if we want to express the transformation rules from the AADL meta-model to the UPPAAL one, the first step is to write our own UPPAAL meta-model. This means that we shall describe the construction rules to which timed and synchronized automata shall conform. In the scope of our study we built the following one.
III. AADL dependability model TO UPPAAL Transformation

A. Generalities

Based on the star tracker example seen before, the goal of this section is to propose a way of building a model of the FDIR using AADL and then transforming the model into UPPAAL. The point here is not to find a definitive solution considering the method objective. What we want to do is to be able to validate the transformation approach theoretically seen before upon a possible way of designing the FDIR. At the end of this section a transformed UPPAAL model shall be generated from the AADL model and some dependability properties shall be validated using the UPPAAL model checker.

B. Modelling using AADL

1. Simplification of the AADL Meta model

As it has been shown before from an idealistic point of view, a space system can be decoupled into functions, a function at a given level N encapsulating the sub-functions of level N-1 recursively. This notion imposes the usage of the structured description language in which this decomposition can be expressed. In AADL, the only component which allows such a hierarchical approach is the system component.

Moreover, a choice remains to be made and concerns the description of the dynamic of the system. We can choose either to use the error model annex or the behaviour annex. Because the error annex is dedicated to the expression of very high level propagation properties and to stochastic analyses (which are not in the scope of our approach) and because it weaves implicit port connections (error propagation) which adds implicit semantic to the model, we choose to describe all the dynamic thanks to the behaviour annex.

So, in order to ease the construction of an AADL model representative for the system to design, the first thing we try to define is a generic pattern which can be reproduced for each functional level. To define this pattern we will first restrict the usage of AADL to the relevant notions for our study. The relevant notions chosen are the following. The associated AADL syntax and shapes can be seen in annex 10:

- **A system**: a composite entity defining an interface (input and output events and data ports)
- **A system implementation**: is relative to a system. It defines
  - The dynamic of the system
  - The subcomponents of the system (subcomponents are instances of other implementations of the system)
  - The connections between the system interface and the interfaces of the systems relative to its subcomponents
- **Event**: can be input or output and is relative to a system interface
- **Behaviour annex**: defines a state machine for a system implementation. The state machines is made of
  - States
  - Transitions linking states. Each transition has
    - a guard (receiving or sending an event, a timeout expiration)
    - an action (computation time)

This is a restriction in the sense that we won’t use all the design possibilities provided by the language in our AADL models. In order to inform the user in case of incorrect constructions (with respect to the transformation capabilities), we will first reduce the AADL metal model to keep only the relevant elements.

In addition, we will not use all the possibilities provided by the behaviour annex. We will use neither the composite and concurrent states nor algebraic facilities. As a consequence we simplify also the meta-model of the annex only to use a simple version of state machines that we will describe later. The simplified version of the AADL meta-model and behaviour annex meta-model can be found in the annex 2 and annex 3.

2. Building a generic composition pattern

The pattern below represents the way we will structure a functional level. It is mainly composed of three sub composite elements:

- The ERROR PROPAGATION MODEL: It is a state machine describing how the functional level (of level N) propagates the errors according to its modes and with respect to the injections of faults that can occur. The notion of modes means that an error coming from a lower level or from a transversal layer may (or not) be propagated. For instance if the Estimation function (level N) computes the attitude from the star trackers 1 and 2 (level N-1) then it is not sensitive to a failure
occurring on the star tracker 3. The state machine describes how to propagate an error, taking into account these modes (this error is said to be at level N)

- The DECISION MODEL: The decision model is a member of the FDIR strategy. It is a state machine in charge of the management of the errors being propagated at the current level and of the management of the detection events coming from the level N-1. A decision is a passive element which doesn’t modify the system (ex: temporisation); it defines what to do in the presence of errors and commands the ACTION MODEL.
- The ACTION MODEL: The action model is a member of the FDIR strategy. It is a state machine in charge of executing the reconfiguration actions ordered by the DECISION MODEL of level N or the ACTION MODEL of level N+1. It constitutes a set of orders that modify the system (ex: reset, switch on/off)

Figure 14 : AADL generic pattern

3. Building generic behaviours

One thing which is interesting with the characteristic pattern is the reusability of components. In fact in AADL systems can be implemented once and instantiated many times. The decomposition we made allows expressing typical behaviours that can be reused in other contexts. However some models are specifics. For instance, the error propagation model is really specific to the functional level being designed. Anyway, it is possible to find some basic and reusable propagation schemes (divergences etc.) that can be representative enough for our test cases. For instance, the final UPPAAL representation\(^1\) of a fault monitoring element can be seen below. This reusable monitoring element is sensitive to the occurrence of an event (error_propagation_event) only when it is active (in st_on state). On the occurrence of this event it transmits the detection event:

Figure 15 : detection pattern (DECISION MODEL)

---

\(^1\) The AADL framework doesn’t provide any graphical tool to represent automata. As a consequence we chose to show the automata in their final shape (after transformation)
In addition, as it has been seen before and considering that FDIR is a hierarchical composition of detection, isolation, and recovery mechanisms, at decision model level, the decision of taking recovery actions of level N are decided. Anyway, these actions may not solve the problem and as a consequence may not stop the initial fault from being propagated. This means that a FDIR block behaves according to the following UML activity diagram:

Figure 16: FDIR block global behaviour

As an example, the “confirmation and reset” FDIR mechanism made at equipment functional level on the occurrence of the star tracker fault, complies clearly with this approach. The final UPPAAL representation can be seen below:

Figure 17: confirmation and reset FDIR block (DECISION MODEL)
On the automaton above the decision model has been built as follow:

- The fault detection is received
- The fault presence is confirmed during a certain amount of time
- If the presence is confirmed, the reset action is ordered
- If the problem is not solved, the block asks the level N+1 for managing the fault

The execution of the reconfigurations is described by the action models. For instance, the UPPAAL automaton below illustrates how a generic action model which manages a classical cold redundancy of two elements among three. On the reception of a given reconfiguration order (coming from the decision model), it switches off or on the corresponding device(s):

![UPPAAL automaton](image)

Figure 18 : cold redundancy 2/3 pattern

At the level of error propagation models, there is no generic aspect except some high level considerations (model of sensor, of a processor etc.) Anyway, the most important thing in these models is that they have to consider that equipment reaches a failure mode when a fault is injected. But, even in the case of a single failure mode (and so a single effect), the “triggers” firing the fault may be different. We consider two types of triggers:

- The first is a transient event. Its characteristic is that we consider that it is a sporadic event that has not destroyed the equipment but just made it temporarily unable to operate normally. A reset of this leads to restore the nominal behaviour (disappearance of the fault)
- The second type is the permanent event. This type of event leads to degrade the behaviour of the equipment permanently (cannot be repaired). A reset of the equipment in such conditions will lead, at restart, to the error being propagated once again. Introducing these two types of fault triggers will helps us at validating different recovery scenario.

In the present case, the error propagation model used for modelling the numerical (sending data on a numerical multiplexed bus) star tracker sensor can be seen in the annex 4.
C. Experiment on sensor fault propagation

At this step we have four inputs, the functional decomposition, the simplified AADL meta-model, the generic construction pattern and a set of generic behaviours. We can now build the model relative to the test case of the star trackers. Due to the hierarchy of AADL diagrams and to the number of event connections, an entire diagram is too complex to be shown directly so we will only describe here the content at each level of the different structural elements seen before.

1. System of level 0: STAR TRACKER
   a) Propagation model
   The propagation model instantiates the sensor pattern of Annex 2
   b) Decision model
   The decision model instantiates the detection pattern of Figure

2. System of level 1: ESTIMATION
   a) Systems of level N-1
   The system contains three instances of the sub system of level 0 star tracker.
   b) Propagation model
   The propagation model describes the divergence of the position computation in case of error. It presents all the execution modes of the estimation function: \(\text{nominal\_using\_sst1\&2}, \text{nominal\_using\_sst2\&3}, \text{nominal\_using\_sst1\&3}, \text{degraded\_using\_sst1}, \text{degraded\_using\_sst2}, \text{degraded\_using\_sst3}, \text{survival}\). In each of these modes, when a visible (i.e.: the star tracker is used in this mode) error is propagated from the star tracker sub function to the estimation, the sub function will enter in a divergent mode and after a certain amount of time the position computation error is propagated if the star tracker failure persists.
   c) Action model
   The action model commands the reset of each of the star tracker.
   d) Decision model
   The decision model instantiates the pattern of Figure 15

3. System of level 2: AOCS MANAGEMENT
   a) Systems of level N-1
   The system contains one instance of the sub system of level 1 estimation.
   b) Propagation model
   The propagation model describes the divergence of the attitude computation in case of error of the position computation made at level N-1. It presents all the execution modes of the AOCS management function: \(\text{nominal}, \text{degraded}, \text{survival}\). In each of these modes, when a visible (i.e.: wrong attitude acquired) error is propagated from the estimation sub function to the AOCS management, this function will enter in a divergent mode and after a certain amount of time the attitude computation error is propagated if the position estimation failure persists.
   c) Action model
   The action model implements the pattern of Figure 13
   d) Decision model
   The decision model is the highest level reconfiguration manager. It implements the detection pattern for monitoring the outgoing attitude. If a divergence exceeding the threshold is detected, the attitude of the spacecraft is no more ensured. A survival mode request is ordered to the ACTION MODEL.
D. Transformation algorithm

At this step we have an AADL model representing our system to be validated. This section is dedicated to the transformation algorithms written in Kermeta and which allow the final UPPAAL model to be generated. The following questions are raised:

- How to translate AADL state machines into UPPAAL timed automata?
- UPPAAL doesn’t allow building a hierarchical structure of the system. So how could we transform a structured view into a non structured view?
- UPPAAL is not able to validate huge models (limit to be found). So how could we express a quantitative criterion for this limit and how to simplify the model in order to make it simple enough?

1. Flow diffusion

The first step of the algorithm is to retrieve the flow paths of all the events within the AADL structure. In fact as we have said there is no hierarchy inside an UPPAAL model. The problem is that in AADL state machines are defined with respect to the interface the system defined. For instance if a system interface defines an input event port names “ev1” one possible state machine for this system is:

![Figure 19: flow diffusion problem 1](image)

At the level of the system there is no information about the sender of the event. As a consequence we need to go through all the architecture in order to associate receivers to their respective sender. Moreover it has to be taken into account that in AADL, systems can be instantiated several times, so the input events are not necessarily connected to the same sender depending on the senders and receivers instances. In addition, AADL manages four types of connections, the 1-1 connections, the 1-N connections, the N-1 connections and the N-N connections. In UPPAAL, the only notions available are “channels” which can be point to point (1-1) or broadcasted (1-N).

Let’s take the following example to illustrate the algorithm of diffusion resolution:

![Figure 20: flow diffusion problem 2](image)

There are four instances of system1 inside a model. All these instances are connected as follows:

![Figure 21: diffusion example](image)
Let’s note $E(i,j)$ the event port associated to the event port $ev(i)$ of the instance $I(j)$, $d(i,j)$ the data sent on the event port $E(i,j)$ and $D(i,j)$ the data $d(i,j)$ decided to be broadcasted.

On the diagram below black points are terminal nodes and white points are non terminal nodes. Each edge is identified by the identifier of the step of the algorithm.

Algorithm inputs: root node $I_0$

- Step 1: from all the terminal nodes, propagate the output event data to the parent node until the root node is reached
- Step 2: in the root node, propagate the outputs events data of first level subcomponents to the inputs event ports of the first level subcomponents.
- Step 3: do, on all subcomponents until the terminal nodes are reached again:
  - Step 3_1: propagate from the input event data of the current node to the input event ports of the first level subcomponents
  - Step 3_2: propagate the outputs events data of first level subcomponents to the inputs event ports of first level subcomponents.
- Step 4: Rebuild the graphs for all terminal nodes according to the propagation
2. **State machine transformation**

In AADL all the dynamic information are carried by the transitions. The transitions allowed for transformation are of the form: 

\[ \text{Source} - \{(\text{on timeout } X) \text{ or (sending event) or (receiving event)}\} - \text{target} \{ \text{computation}(X,Y) \text{ or delay}(X,Y) \text{ or (sending event)} \}; \]

Where \([-\rightarrow\) represents the guard field and \(\{\) represents the action field. The transformation shall turn this pattern into the notion supported by UPPAAL:

- **Transition guard:** condition that shall be true to fire the transition
- **Location invariant:** condition that shall be true to stay in the node
- **Transition synchronization:** one sending event or one receiving event
- **Transition update:** update of variables after the transition has been fired.

Today the transformation algorithm supports the following transitions (x is the local clock of the UPPAAL template). UPPAAL limits the number of synchronization events per transition to 1. There is no such limitation in AADL. For the purpose of the transformation we chose the more restrictive modelling solution (the UPPAAL one):

- **a) Send event**
  
  Source \(\rightarrow\) Target

- **b) Receive event**
  
  Source \(\rightarrow\) Target

- **c) Timeout**
  
  Source \(\rightarrow\) Target

- **d) Computation 1**
  
  Source \(\rightarrow\) Target \{computation (20, 30) \}
  
  Target \(\rightarrow\) End

- **e) Computation 2**
  
  Source \(\rightarrow\) Target \{computation (20, 30) \}
  
  Target \(\rightarrow\) End

3. **User interface**

As it can be seen when computing the events propagation, only complete paths are built. This is a problem considering that all the information is not in the model. In fact, the AADL model goal is to describe what can do the system. It doesn’t take into account validation aspects (chosen scenario etc.). In other words fault injection scenarios are not parts of the AADL model but parts of a dedicated validation model. The transformation shall as a consequence provide a way to plug together the fault injection model and the transformed model.
To inform the transformation algorithm that some events are controlled by another model it is possible to annotate the model thanks to the AADL properties. A given input event port can be enriched as follows:

![Figure 27: model annotation](image)

**E. Model validation**

At the end of the construction of the transformation algorithm written in kermet, the AADL model can be transformed. This section will illustrate some results that we have observed. All the experiments are based on the following fault injections model and concern at the moment the permanent faults:

- 
  • _6_USER_PERMANENT_FAULT: injection of a fault in the star tracker 1
  • _9_USER_PERMANENT_FAULT: injection of a fault in the star tracker 2
  • _12_USER_PERMANENT_FAULT: injection of a fault in the star tracker 3

![Figure 28: fault injection model](image)

When simulating the model transformed, we observed that the expected behaviour was respected for all the cases we try to performed. We try to prove some properties thanks to the expression of them into the LTL form.

1. **Property of good reconfiguration**

   “It is always true that in case of a single failure, if the star tracker 1 if off (i.e. after reconfiguration) and the star tracker 2 and 3 are on then the failure that had occurred is the failure of the star tracker 1”

The corresponding LTL form of this sentence is:
2. Property of error propagation

“It is always true that in case of a single failure, the degraded and survival AOCS modes are never reached”

The corresponding LTL form of this sentence is:

\[
A[] ( ( \text{inst\_USER.i} == 1 ) \implies \neg \text{inst\_15\_AOCS\_PROPAGATION\_MODEL.st\_degraded\_mode} \land \neg \text{inst\_15\_AOCS\_PROPAGATION\_MODEL.st\_survival\_mode} )
\]

“It is always true that in case of a two single failures, the survival AOCS modes are never reached”

The corresponding LTL form of this sentence is:

\[
A[] ( ( \text{inst\_USER.i} == 2 ) \implies \neg \text{inst\_15\_AOCS\_PROPAGATION\_MODEL.st\_survival\_mode} )
\]

“It is always true that if the AOCS is in survival model then 3 single failures had occurred on the system”

The corresponding LTL form of this sentence is:

\[
A[] ( \text{inst\_15\_AOCS\_PROPAGATION\_MODEL.st\_survival\_mode} \implies ( \text{inst\_USER.i} == 3 ) )
\]

3. Counter Example

The model generated is also able to detect wrong configuration of the FDIR strategy. For instance we can try to prove that:

“It is always true that in case of a single failure, the estimation of the attitude never goes out of bounds”

The corresponding LTL form of this sentence is:

\[
A[] ( \text{inst\_USER.i} == 1 \implies \neg \text{inst\_3\_ELF\_ESTIMATION\_PROPAGATION\_MODEL.st\_quat\_computation\_out\_of\_bounds\_fm} )
\]

This property is verified on our model. Anyway, if the FDIR strategy is well chosen, the properties are verified because the reconfiguration occurred before the attitude estimation becomes wrong. As a consequence, if, for instance, we change the detection confirmation time in the FDIR strategy such that “confirmation time > delay to compute a wrong attitude” then the property won’t be verified anymore. In such a case, the UPPAAAL tool gives us a counter example. This last shows that the position estimation has gone out of the bounds and the estimation error has been propagated before the confirmation phase ends.

![Figure 29: counter example](image)

F. Experiment on communication buses

After the method illustration above, we now focus on more complex cases of fault propagation and of common cause failures, illustrated on the case of communications buses. What has not been considered in the previous sections is that the star trackers communicate with the estimation sub function thanks to a 1553 multiplexed bus. The aim of this section is to propose a model based implementation for this bus describing how faults and failures propagate and how they are captured and recovered.

1. Functional decomposition

In term of functions, the onboard implementation can be seen as follow:

- Each client (ex: sensor) of the 1553 bus is connected to the bus thanks to a bus terminal.
The onboard computer embeds a 1553 bus controller which is in charge of managing the real time traffic.

The bus controller has a static scheduling table in which the periodic requests are described:
- When a data has to be acquired, the controller formats a specific asking frame to the equipment and sends it to its terminal.
- On reception, the sensor formats and sends the data.
- On reception of the data, the controller sends back an acknowledgement to the equipment.

For the purpose of the model we use the following functional decomposition to illustrate this behaviour. The onboard communication management (at application level) is split in two parts: the management of the avionic buses (used by the AOCS) and the management of the payload buses. In the example we only focus on the avionic buses. The avionic bus management (at equipment function level) is in charge of implementing the bus controller (for the avionic bus) and finally at equipment level we find the hardware buses. Though they are members of the 1553 bus sub system, we consider that the terminals are located, in the functional decomposition, in the scope of the equipment they are used by. This solution will simplify the AADL model.

![Figure 30: onboard communication, functional decomposition](image)

### 2. FDIR strategy

The architecture chosen for the AOCS usage is composed of two 1553 buses which are in cold redundancy. In order to be confronted to ambiguous cases we will consider that the terminals are not able to monitor the bus and cannot notify any failure. As a consequence all the monitoring abilities will be concentrated on the controller side. For the purpose of the example we will consider the following strategy:

- **At equipment level:** N.A.
- **At equipment function level:** any not responding equipment is detected. This detection is confirmed during a certain amount of time in order to simulate the robustness of the protocol (explicit retries etc...). If the failure persists a confirmation event is sent to the upper layer.
- **At application level:** the confirmation event is acquired and the sequence of confirmations events is analysed. The idea of this analysis is to find a pattern which is characteristic of a bus failure. In our case we say that a burst of events (succession of events appearing during a short period) characterises a bus failure. If a characteristic sequence is found then the bus failure confirmation is sent to the upper layer.
- **At system level:** The bus failure confirmation is acquired and the reconfiguration starts:
  - All the bus terminals are ordered to switch to the bus 1553 side B
  - All the equipments connected to the bus perform a restart.

### 3. Problems

In the scope of this second case we consider neither the terminals failures nor controller failures. Only the bus failures are considered. The 1553 bus is plugged between the start trackers and the estimation function seen before. Each star tracker will be connected to the bus thanks to a bus terminal. Anyway considering error propagation the following problems are raised.
a) Isolation capability

Let’s consider a star tracker failure mode for which, from the estimation sub function point of view, a failure is visible and says “sensor mute”. The failure may be the result of:
- An internal error of the star tracker
- An internal error of the 1553 bus (side A) which stops the transmissions from flowing

As we consider that the terminals are not able to monitor the bus and as the estimation function computes the position from two star trackers, the internal error of the bus is seen as simultaneous failures of the two star trackers in use (in nominal mode). As a consequence, in absence of bus FDIR mechanisms, the AOCS FDIR strategy is applied (confirmation, reset and reconfiguration) and won’t solve the problem.

b) Hidden faults

If we consider that the bus fails and is not reconfigured, the communication from the sensors to the OBC is lost. As a consequence if during the silence time interval the sensor fails, this cannot be seen and will be revealed only after the bus reconfiguration. In such a case, the fault of the sensor is hidden by the bus fault.

c) Dormant faults

The dormant faults have to be distinguished from the hidden faults as follows:
- A hidden fault cannot be seen by the service consumer because of the presence of another fault
- A dormant fault cannot be seen by the service consumer because it doesn’t use the service yet

One case of dormant fault is the case of a not pertinent reconfiguration. Let’s consider that no FDIR strategy is implemented at bus level. As we have said, if the bus fails, the AOCS will see a double failure of the two star trackers and according to its strategy, it enters the degraded mode and tries to wake up the cold redundant star tracker. However, the star tracker is not able to respond because of the bus error. As a consequence the survival mode will be ordered.

4. Modelling bus FDIR

In our example we consider the bus as our target for fault injections. The model is very simple and its goal is only to propagate the errors to the bus terminals

Figure 31: bus error propagation model

At the bus terminal level we have to consider the failure visibility problems seen before. In other words the following cases have to be taken into account:
- The terminal allows the propagation of the sensor internal error “sensor mute” in absence of bus failure
- The terminal propagates the error “sensor mute” on a bus failure occurrence.
- The terminal propagates the “sensor mute” at start up when the active bus is failed.

All of these propagations have to consider the notion of time. In fact, the propagation won’t occur if the bus controller doesn’t ask for the data. To simulate this aspect, we take the hypothesis that the transmissions are periodic. As a consequence, an error can be propagated within the request interval [0, Period]. A generic pattern for error propagation on double cold redundant buses has been built and can be seen in annex 5.

On the controller side the goal is to confirm the not responding sensors during a short period. As a consequence we built a generic confirmation pattern instantiated three times (on per sensor) at avionic bus management level. At application level (onboard communication management) confirmed not responding sensors have to be analysed to see if the sequence of events appears during a short period of time. In other words the FDIR at this level searches for the detection of a failure burst. For the example, as we have at most two equipments we limit the sequence length to two. At satellite management level (system level), the reconfiguration occurs on reception of the confirmation event coming from the onboard communication level. The reconfiguration asks for the bus terminals to switch on the cold redundant bus and all the sensors are asked to perform a reset. The following figure shows the burst detection pattern built, others are presented in the annex 7.
5. Validation

For the validation we use a simple user model allowing injecting one permanent fault on each of the two buses.

a) Simple bus failure

A first test case is the injection of a single permanent fault on the 1553 bus side A. At AOCS level, the two sensors currently used will become mute and the estimation attitude starts diverging. The expected behaviour is the following:

- The “sensor mute” events are detected by the avionic bus management function
- These events are confirmed during a certain amount of time
- After the confirmation, if the failure persists, the cascade of events are analysed by the onboard communication management function.
- The cascade is interpreted to be characteristic of a bus failure and the reconfiguration occurs at satellite level.
- All the bus terminals are requested to switch to the side B and all the client equipments are restarted.
- The position estimation is recovered and the degraded AOCS mode is not ordered by the AOCS FDIR.
This behaviour has been successfully observed by simulation and then we propagation property has been proven by model checking:

“It is always true that in case of a single failure, the degraded and survival AOCS modes are never reached”

A[0] ( ( inst_USER.i == 1 ) imply not inst__20_AOCS_PROPAGATION_MODEL.st_degraded_mode and not inst__20_AOCS_PROPAGATION_MODEL.st_survival_mode)*

b) Double bus failure

The second case goal is to illustrate the problem of latent failures. In order to observe the behaviour of the system in such a case, after a first well reconfigured failure we inject a fault on the 1553 bus side B. Theoretically, the second case is the same as the first one except that at satellite management level, the survival mode is ordered to the AOCS and to the on board communication management.

Anyway, in our case we modify slightly this scenario. In fact, in our architecture, the satellite management level doesn’t order the AOCS to go to the survival mode. As a result on the second bus failure occurrence the context is the following:

- The ACOS is in nominal mode
- Two sensors become mute
- The off sensor won’t be able to transmit its data at wake up (no 1553 bus available)

As a consequence the expected behaviour of the ACOS FDIR is:

- Each failure on the sensors is confirmed in time
- Each sensor is restarted
- The two failures are confirmed after reset and the ACOS start reconfiguring to the degraded mode:
  - The two sensors are turned off
  - The off sensor is woken up
- At start-up the last sensor becomes mute
- The failure on the sensor is confirmed in time
- The sensor is restarted
- The failure is confirmed after reset and the ACOS is reconfigured in the survival mode

This behaviour has been successfully observed by simulation and then the property of correct configuration has been verified by model checking:

“It is always true that in case of one failure on each bus, the satellite management and the ACOS management function are torn to the survival mode”

A[0] ( ( inst__20_AOCS_PROPAGATION_MODEL.st_survival_mode and inst__30_SATELLITE_DECISION_MODEL.st_safe) imply (inst_USER.i == 2) )

G. Conclusion

At the end of this first part of study we successfully built an AADL model describing an error propagation model and its associated FDIR strategy for a sub branch of a space system. Up to this stage we did not encounter combinatorial explosion problem in the UPPAAL model and the properties where successfully demonstrated.

As a major drawback we can say that UPPAAL timed automata restrict a lot the possibility of the AADL behaviour annex but advanced transformation algorithms may solve the problem. Anyway, the modelling activity looks like designing a structured UPPAAL model instead of transforming the AADL semantic into the UPPAAL one. However, what looks interesting is that we gathered the dynamic of the system in conformance with the AADL meta-model which is a good approach in terms of abstraction. In fact, we have a structured view of the dynamic of the system written in a language that is transformable in other languages. This could not have been possible if we had directly written it in UPPAAL for instance. We made the experimentation with UPPAAL but we hope other tools could be used and complete the analysis.

In addition, modelling in AADL has the advantage to decouple the specification of the FDIR strategy from the implementation that can be done. From the model, the requirements can be extracted but no constraint on the software conception or coding techniques is imposed.
IV. Dependability/FDIR modelling method

A. Discussion of the current modelling method

At the end of the first part we have modelled thanks to a simple modelling technique both faults and failures propagations and FDIR representative strategies. Moreover, we were able to demonstrate some properties by model checking considering temporal aspects. In that sense the initial objectives are met. Anyway our model suffers from some limitations considering the potential usage of such a method:

- Maintainability: AADL is a functional description language which expresses how a given system is structured and behaves. An AADL model is also often designed in order to check some properties concerning the performances of the system such as flow latencies or resource consumptions (engineering model). In the contrary, the model previously designed is a dependability/FDIR dedicated model. In fact, the fault and failure propagation are studied with respect to the functional design. As a consequence, our model integrates the functional design from the faults and failures point of view. Considering that there is an engineering model, there is a problem of information redundancy. The two models (engineering and dependability/FDIR) shall be manually kept consistent in the case of successive architecture modifications. If for instance, a physical bus is removed from the design, in the dependability model, some fault and failure propagation may be lost (or modified).

- Complexity and reusability: The compositional approach we choose adds a lot of complexity inside the model when dealing with transversal faults and failures propagation. We pointed out this problem when we designed the 1553 bus. In fact, the model complexity in terms of number of connections becomes very rapidly unmanageable. Moreover, in the previous model, the dependability concepts of faults, errors, failures and failure modes are gathered all together inside the error propagation models. It causes a lot of problems of scalability. Indeed, the more failure modes we add in the error propagation model the more it becomes complex. This is normal and unavoidable. But as we shall consider the dynamic of the system and especially the state of components (switch off/on etc.) and software (initialization, nominal etc.) are, the dormant and hidden faults become really hard to manage; the model having to deal with all the combinatorial of all the new faults injected and all the present faults either if they are active, hidden, or dormant. The fault and failure multiplications cause also the propagation model to become really specific (not reusable) to given design element. In addition, the dependability model we designed doesn’t take into account explicitly hardware and software components. In other words, they are not parts of the model. They are analyzed, and their interactions are captured into faults and failures propagation as well as their way of failing is expressed in the Error propagation models. This means that except the final proving method, the modelling technique doesn’t provide any support to the analysis phase.

- Understanding: The dependability model we designed manages the error propagations thanks to event connections. Once again, AADL expresses functional notion and errors propagation are dysfunctional notions. This means that such propagations cannot appear, with respect to the good usage of AADL, in the form of functional notions. The confusion, in addition to the complexity of the design, may lead to understanding problems.

After these observations, it looks obvious that we need to enhance our modelling techniques in order to find a more applicable method. Anyway, the possible directions to follow are emerging:

- Decoupling engineering, dependability and FDIR models: In order to avoid information redundancy, a better approach seems to build a method converging to a single full model integrating both functional, dependability and FDIR aspects. Moreover, we can think about a simple incremental design method:
  - The engineering model is designed to express the structural and behavioural aspects.
  - The dependability model is plugged on the functional model and describes the way of components fail.
  - The FDIR model is plugged in the previous merged model in order to describe the way faults and failures are managed.

- Decoupling dependability concepts: In order to avoid the complexity and reusability problems, we can think to decouple faults, errors and failures notions (and their associated dynamic). This means that instead of a full propagation model, a way of describing the process leading fault to failure within a component thanks to dedicated smaller models connected in a systematic manner may ease the dependability layer design.

- Propagation deduction: As it has been said, the functional model carries implicit faults and failures propagations paths. Plugging the dependability model onto the functional model makes this implicit propagation accessible (the transformation algorithm will process the whole model). As a consequence, explicit event connection expressing propagation becomes useless. This may reduce both complexity and understanding problem.

Expressing these possible directions leads us to conclude about the necessity of building a layered model, each layer being in charge of a specialized actor of the design. The three layered chosen are summarized below:
Moreover, this new approach also leads to think about what will be expected about the transformation to UPPAAL. In fact, if the propagation shall be extracted from the functional model, this means that the model shall be analyzed, the semantic understood and propagation deduced. Furthermore, as faults and failures propagation don’t appear in the model, this means that they shall be calculated based on propagation rules. This last point seems very close to the notions seen with the error model annex.

### B. Decoupling dependability notions

Thanks to basic concepts and terminology of dependability defined in the bibliography [8] what are called the “dependability threats” may be described as a recursive causal chain from faults to failures through the notion propagation. A propagation chain looks as follows:

Fault $\rightarrow$ Error $\rightarrow$ Failure $\rightarrow$ Fault $\rightarrow$ Error $\rightarrow$ Failure ... $\rightarrow$ Fault $\rightarrow$ Error $\rightarrow$ Failure

This chain is the basis we propose for modelling the dependability layer. In fact, a “Fault $\rightarrow$ Error $\rightarrow$ Failure” pattern is related to one component. This introduces the notion of point of view. The inputs of a component are faults and the outputs are failures, a failure being linked to a provided service and it becomes a fault from the service consumer point of view when this service is lost or corrupted.

Besides, inside a given component, a fault can be seen as a causal phenomenon which, when activated, leads to the creation of an error (or several). It is important to think about the fault in term of phenomenon, it is not necessarily an instantaneous event. It could be a situation or a more complex process. As a consequence, the notion of fault comes with the notion of fault dynamic. However, the notion of fault activation will be defined in the next sections.

As said above, submitted to this causal phenomenon an error is created. An error can be seen as temporal process (can be instantaneous also) which describes the dynamical process from the cause to a potential deviation of the provided service i.e., a failure. The level of abstraction has to be well chosen according to the studies which have to make. Anyway, the process relevant for an error may alter a given functional service. If the service can no longer be provided, the failure is fired. In other words, an error may make the component switching from a nominal mode to a failure mode. The transition between the two modes engenders the failure event to be fired.

Conversely, switching back from a failure mode is not necessarily linked to the disappearance of the causal phenomenon. It could also depend on the error nature, which may not alter the service in a permanent manner. The system may also enter in a behaviour mode where it is no longer sensitive to the presence of the fault and the errors it has created. In addition, the notion of failure modes is not as simple as a set of modes reachable from a nominal mode. It could exist a hierarchy of failure modes. For instance the mode "loss of one output signal" is included in another "lost of two outputs signal". A failure mode cannot also always be seen as a state in an automaton. For instance the failure mode "state of charge corrupted" for a battery introduces the fact that the state of charge can be corrupted at 75%, 80% etc. depending on a continuous process. In such cases the notion of failure trajectory is often used, but we will not consider this point in our study. Finally, as it is a chain, the notion of root faults and terminal failures shall be extracted. Root faults are the events which trigger the starting of the propagation while terminal failures are linked to the lost or corrupted services of the highest levels.

#### 1. Modelling fault dynamic

In order to model the notion of faults, we have to think about the attributes of the causal phenomenon seen before. First, we don’t focus only on the propagation in the sense of the fault presence. In fact, as we are interested in the recovery process we also have to deal with the disappearance of the faults and as a consequence with the process leading a fault to disappear after its appearance. Moreover, a fault could be active, dormant on hidden. The “hidden” notion shall not be considered when building the model of a fault dynamic. In fact, it is an active fault masked by another active fault inside the system. Anyway, to express the notion of active and dormant fault let’s take the following example:

A software part is acquiring a data on one of its input ports while it behaves in nominal mode. This software has three modes:
- An idle mode in which the software sleeps
- An initialization mode in which the software enters when activated by the execution platform.
- An nominal mode is entered if the initialization mode is successfully processed
In such a case, considering a permanent fault, the system is submitted to the fault only when the software is in nominal mode, otherwise the fault is dormant. This means that independently from the presence of the fault, there is an interaction between the functional behaviour of the component and the possible faults of this component. As a consequence, from a modelling point of view, the phenomenon linked to a fault is active when the component could be sensitive to the fault and dormant otherwise.

The automaton below shows a typical model of root fault dynamic which has been used. It manages the notions seen above and two kinds of event leading the fault event to appear (transient and permanent). These two events have been implemented in order to control the fault injection, but stochastic properties can be added to them in order to perform further analyses. Moreover, it is not the only fault dynamic model used. Other models have been built in order to deal, for instance, with living duration of faults or to express the chained fault (faults which are fired in the presence of a failure of the service producer).

Figure 34: root fault dynamic model

2. Modelling error dynamic

After having modelled faults, we have to model the way they create the errors leading to enter the failure modes. There are clearly two types of models for error modelling. A model enclosing the nominal and failure modes and a set of several error models which interact with the modes model to fire failures. For modelling efficiency reasons we chose to merge these two models types into an approximation which is called the fault propagation model. In this merged model, an error is considered as a process leading from a nominal mode to a failure mode. Moreover, this model shall also be connected to the faults models. When changing of mode, it will have to define (event sending) to which faults it is sensitive or not. The automaton below shows a simple fault propagation model of a processor.

Figure 35: Error dynamic model
3. Modelling failure dynamic

The model of failure dynamic is simpler than previous ones. The only thing which has to be kept in mind is that failures, as faults, can appear and disappear. In the modelling technique we gathered these two notions inside a single model which is shown below:

![Failure dynamic model](image)

**Figure 36: Failure dynamic model**

C. Propagation chain pattern

At this step we have decoupled the dependability notions. They can be now, bound together in order to build a dependability chain which will be used in the models of next sections. However, as the number of faults and failures grows, the fault propagation model will become very complex. In order to solve this problem we allowed building composite fault propagation models in order to be able to manage faults and failures independently if needed. Moreover, the dynamic of faults (active, dormant) are linked to the dynamic of the system. This means that in terms of propagation, a consistent behaviour of the system shall be maintained. In other words, the fault propagation model shall be interfaced with the functional side in order the changing of modes to be representative. In addition, these interfaces are not necessary relative to hardware or software interfaces (reset signal, timeout etc.) They can also be linked to environment constraints. For instance, a solar panel switch from the mode “eclipse” to the mode “sun present” according to the sun visibility. This means that the fault propagation model shall also be interfaced with the environment. The diagram below summarizes the dependability pattern.

![Dependability pattern](image)

**Figure 37: dependability pattern**
D. Fault and failure compatibility

At this step the propagation pattern has been designed. The problem is now to find a method to connect all the patterns together inside a consistent model without explicit connections and to connect the pattern to the functional model. These two points are very close to the potential of the error model annex seen before. Though it could have been possible to build our model thanks to this annex, we chose anyway, to continue using the behaviour one. In fact, we found, that compulsory propagation stopped or not by using guards, is a too complex process in the scope of our study and that a simpler modelling technique is expected. As a consequence in this section we define our own semantic concerning the propagation rules.

1. Deduced failures propagation

As it has been seen, the dependability analysis is based on a functional model which is expected to be complete. This functional model can help at deducing faults and failures propagation. This section summarizes the deducible rules. In the diagrams below the fault models are denoted \( f \), the fault propagation models \( fm \) and the failure modes \( F \) and the implicit propagations are expressed in red dash lines.

a) Software/Software dependencies

If two components are connected by a functional dependency, a component sending a data to another (or several)), then a failure related to the alteration of the output data could propagate to all the consumers of this data.

![Figure 38: software/software dependencies](image)

b) Hardware/Software dependencies

If a software component exploits a hardware device for computing, storing etc. a failure related to an alteration of the service required from the hardware device could propagate to all the software components which exploit this hardware device.

![Figure 39: hardware/software dependencies](image)
c) **Hardware/Hardware dependencies**

If hardware components are connected thanks to a physical medium then a failure of a hardware component could propagate to the other hardware components connected to this medium.

![Figure 40: hardware/hardware dependencies 1](image1)

Moreover, the physical medium is itself a hardware component. As a consequence, a failure of the physical medium could propagate to the hardware components connected to this medium.

![Figure 41: hardware/hardware dependencies 2](image2)

2. **Fault/Failure classification**

a) **Constraints**

Though we are now able to determine how faults and failures propagate, these propagation rules are not sufficient to connect failures to faults. For instance, if there is a failure on a particular RAM segment, this failure won’t propagate to a software component which doesn’t depend on this segment. As a consequence, the propagation rules aren’t sufficient, and we introduce the notion of failure/fault compatibility. Moreover, several failures can propagate according to the same propagation rule but on the failure characteristics depend the fault to which it shall be related. For instance, “corrupted input data 1” and “lost data 1” are two faults related to two different failures, though they propagate according to the same data path. The two following notions become important:

- A fault candidate is a fault to which a failure could propagate according to a propagation rule
- A compatible fault is a fault candidate which matches the failure properties (which are defined later)

Additively, the compatibility between failures and faults shall conserve the dependability patterns used as independent as possible. In other words, the characteristics of a failure shall not take abusive hypotheses concerning the ability of the propagation target to be compatible. For instance, a failure saying “corrupted, detectable data 1” is meaningless, because it
implicitly requires from the receiver side to be able to detect it. As a consequence, the independence between dependability patterns will be lost. To solve this problem we choose to implement the notion of compatibility thanks to extensible mechanisms. The idea is to give as much information as possible on the failure in order not to presume about the receivers capabilities. The previous example will be translated into “corrupted data 1 greater than 3”. In this case, if the receiver monitors the faults thanks to a threshold of 3, it will detect the fault.

b) Used classification

In order to express the compatibility we choose to build a classification which can be attached both to the faults and to the failures. This classification has been build in order to be extensible according to the designers needs. The classification used for the design we made is presented below. This classification is not complete, but sufficient to illustrate our test cases and could be extended.

<table>
<thead>
<tr>
<th>Fault/Failure class</th>
<th>Fault/Failure type</th>
<th>Fault/Failure sub-type1</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Blocked</td>
<td></td>
<td>The data value becomes invariant</td>
</tr>
<tr>
<td></td>
<td>Syntax</td>
<td></td>
<td>The data is not well formed</td>
</tr>
<tr>
<td></td>
<td>Content</td>
<td></td>
<td>The data value is incorrect</td>
</tr>
<tr>
<td></td>
<td>Absent</td>
<td></td>
<td>The data is absent</td>
</tr>
<tr>
<td></td>
<td>Spurious</td>
<td></td>
<td>The data is transmitted abusively</td>
</tr>
<tr>
<td></td>
<td>Early</td>
<td></td>
<td>The data is transmitted too early</td>
</tr>
<tr>
<td></td>
<td>Late</td>
<td></td>
<td>The data is transmitted too late</td>
</tr>
<tr>
<td></td>
<td>Frequency</td>
<td></td>
<td>The data transmission frequency is incorrect</td>
</tr>
<tr>
<td>Resource</td>
<td>Processor</td>
<td>Computing</td>
<td>The computing capability of the processing unit is lost</td>
</tr>
<tr>
<td></td>
<td>Bus</td>
<td>Transporting</td>
<td>The medium cannot transmit data anymore</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>Reading</td>
<td>The memory cannot be read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Writing</td>
<td>The memory cannot be written</td>
</tr>
<tr>
<td></td>
<td>Power Supply</td>
<td>Corrupted State of charge</td>
<td>The state of charge of an energy storing device is corrupted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lost</td>
<td>The power supply capability of energy provider is lost</td>
</tr>
<tr>
<td>Function-centred</td>
<td>Bug</td>
<td></td>
<td>Software bug</td>
</tr>
</tbody>
</table>

All the classification items above were expressed under the form of AADL properties which can be applied on faults and failure. The full AADL property set can be seen in annex 8.

E. Binding functional, dependability and FDIR models

At this step we have a full dependability pattern where faults and failures can be classified in order to express their compatibility. This section is dedicated to the methods used in order to implement the layered model presented earlier in this section. The modelling technique chosen allowed us to build some libraries of commonly used dependability blocks (faults models, failures models, divergence error models etc.). The exhaustive list of the components built is shown in the annex 7.

For the purpose of this section we will take the example of two software processes hosted on two processors. The process A sends a data to the process B. The terminal failure in the system is linked to the data which B is supposed to send to someone and the root fault is an internal fault of the first processor. The fault monitoring resides in the process B and detects the absence of the data coming from the process A. Starting from the functional AADL design below, we first implement the dependability pattern for all the concerned components, choose the faults and failures classification and finally implement the FDIR monitoring.

![Figure 42 : functional design example](image-url)
1. **Step 1: Build dependability patterns**

The first step is to design the dependability pattern for all the components. As in AADL there is only one composite element (the “system” component) we gather inside system elements all the faults, failures and fault propagation models related the each component. First, the processor “proc1” will be the target for fault injection. This means that a root fault will be attached to it (rf0). On the appearance of this fault, its fault propagation model (fm0) fires a failure expressing that the processor has lost its computing capability (F0). This failure is classified: {Resource, Processor, and Computing}. The following AADL diagram shows how it has been implemented.

![Figure 43: processor dependability model example](image)

On the process A side, the failure F0 is compatible with the fault f1 attached to the dependability model of the process. This expresses that the process A is sensitive to the loss of the processor computing capability. The fault propagation model of the process a (fm1), answering to the appearance of this fault fires the failure F1 indicating that the data at the output of the process is lost. This failure is classified: {data, absent}. f1, fm1 and F1 are gathered into an AADL system component as follows:

![Figure 44: process A dependability model example](image)

The dependability model of process B contains a single fault f2 which is compatible with the failure F1. In addition as the failure of process B cannot propagate anymore in our example the failure of process B is a terminal failure (TF2). The fault propagation model (fm2) ensures the propagation within the component from the fault f2 to the failure TF2. The associated dependability model is shown below:
2. Step 2: Binding models

At this step, we have to bind the dependability models built to the functional model seen on the “Figure 42 : functional design example”. In order to construct such a binding we formalized through an “AADL properties set” a set of dependability properties for this purpose. The first property used aims at attaching a dependability model to a functional component. This property is of the form:

\[ \text{PropFailureModelBinding: list of reference (system) applies to (thread, processor, memory, bus, device, process, system);} \]

Admitting, that dependability models are named respectively \(dm0\), \(dm1\) and \(dm2\), adding the following properties in the system enclosing the functional model is sufficient to express the attachment:

\[ \text{PropFailureModelBinding }\Rightarrow \text{ reference dm0 applies to proc1;} \]
\[ \text{PropFailureModelBinding }\Rightarrow \text{ reference dm1 applies to A;} \]
\[ \text{PropFailureModelBinding }\Rightarrow \text{ reference dm2 applies to B;} \]

In addition, though the software to hardware mapping is sufficient to deduce fault candidates, when faults and failures are linked to data transmission, a directive is missing. For instance, if instead of data 1, the process A sends data 1 and data 2, then it is impossible to deduce if the failure \(F1\) belongs to data 1 or to data 2. As a consequence faults and failures have to be attached to the relevant data they belong to. This can be done as follow:

\[ \text{FailureBinding }\Rightarrow \text{ reference dm1.F1 applies to A.data1;} \]
\[ \text{FaultBinding }\Rightarrow \text{ reference dm2.f2 applies to B.data1;} \]

The last step consists in binding the capability for of FDIR mechanism to monitor the fault \(f2\). In the example the monitoring mechanism takes the form of a thread inside the process B and hosted by the processor proc2. The monitoring binding can be implemented, once again, using a dedicated property:

\[ \text{FaultMonitoringBinding }\Rightarrow \text{ reference B.fdir_monitor applies to B.f2;} \]

The result of the modelling activity is summarized on the figure below:
F. Model transformation

1. Algorithm chain

The modelling technique is now complete and the last step before starting to validate using model checking is to transform the model into an UPPAAL one. Compared to the first model we designed, the new model is very complex. First, all (or near) the AADL language is used, this means that the transformation must be able to express transformation rules over all the AADL meta-model, including complex elements, hardware/software mapping, bus accesses and properties analysis. Then, all the faults and failures propagations are implicitly expressed according to propagation rules. This means that the transformation shall solve the implicit connections and explicit them to build the final UPPAAL model.

In order to solve all these problems we built a full object oriented Kermeta program which based on the following algorithm chain:

![Figure 47: model transformation algorithms](image)

For the last two steps of the algorithm chain, the flow resolution (seen earlier) and the UPPAAL meta-model treatments are conserved (slightly modified). This section as a consequence deals, mainly, with the first two steps.

2. Model extraction

The first step of the algorithm chain consists in transforming the model in input (which conforms to the AADL meta-model) into a simpler one. In fact, the AADL meta-model, is very complex to exploit natively, it is split in several meta-models and the structure is not compatible with the usage expected here. For instance, for the purpose of the transformation, the inheritance specificities of each component (process, bus etc) are not relevant. What we are interesting in are the elements, their connections and the properties attached to them. In addition we want the model to be parsed easily. As a consequence, the first step of the algorithm consists in exporting the AADL model into a simpler one compliant with the following data structure:

![Figure 48: Simple AADL model – class diagram extract](image)
The notion of AADL bindings above represents the properties which could be associated to a node, a link or a port. We do not represent the entire class diagram here, but all the components of the model inherit from a root object called CSaadlItem. As a consequence, a binding is characterised by a type of binding, an item on which it is applied and a set of items which are its targets.

[Diagram of CSaadlBinding and CSaadlItem]

Moreover, it can be seen on the previous figure that nodes are expressed as a recursive structure. This means that the AADL hierarchy, at this step of the algorithm, is preserved. We could have chosen, directly, to break this hierarchy because UPPAAL doesn’t manage this notion but we chose to conserve it. In fact, at this step, we haven’t yet solved the dependability compatibility and considering future propagation rules, the AADL hierarchy could be useful. For instance, it is possible to imagine that components gathered into a composite element could express the common physical location (in space) for these components. As a consequence, when dealing with thermal faults propagation the hierarchy might be part of a propagation rule.

So, as the hierarchy of nodes is preserved, the hierarchy of connections linking these nodes shall be preserved. Those links (CSaadlLink) are also expressed as a recursive structure as it is shown below:

[Diagram of CSaadlLink and CSaadPort]

An example of links hierarchy storage is shown below:

<table>
<thead>
<tr>
<th>Port</th>
<th>Previous links</th>
<th>Next links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process1.soft.data1</td>
<td>{}</td>
<td>{C1}</td>
</tr>
<tr>
<td>Process1.data1</td>
<td>{C1}</td>
<td>{C2,C3}</td>
</tr>
<tr>
<td>Process2.data1</td>
<td>{C2}</td>
<td>{C5}</td>
</tr>
<tr>
<td>Process3.data1</td>
<td>{C3}</td>
<td>{C4}</td>
</tr>
<tr>
<td>Process2.soft.data1</td>
<td>{C5}</td>
<td>{}</td>
</tr>
<tr>
<td>Process3.soft.data1</td>
<td>{C4}</td>
<td>{}</td>
</tr>
</tbody>
</table>
3. **Solving fault/failure compatibility**

Solving implicit dependability connections is a more complex problem. In fact, it looks like reversing the modelling technique. The idea is to start from the failures and to explore the model until a propagation rule is matched. Then fault candidates are elected and from the candidates, compatible faults are selected and their effective connections restored. The step by step algorithms implemented in the AADL to UPPAAL transformation program are depicted based on the different situations met. In the next paragraphs identifiers of the algorithm step are annotated on the figures and depicted below them.

![Diagram](image)

**Figure 52: hardware/software compatibility**

- **Step (1):** For all the failures related to an hardware component (F1), retrieve the dependability model to which it belongs (dm1).
- **Step (2):** Retrieve the hardware components to which is attached the dependability model.
- **Step (3):** Add to the targeted components set all the target of all the AADL bindings relative to a hardware/software bindings. Targeted components set = {soft1, soft2, soft3}
- **Step (4):** Add to the fault candidates set, all the faults related to the dependability model attached to each element of the targeted component set. Fault candidates set = {f21, f22, f3, f41, f42}
- **Step (5):** For all the elements in the fault candidates set, compare classification attributes (between failure and fault) and elect compatible faults.

![Diagram](image)

**Figure 53: hardware/hardware compatibility**
• **Step (1):** For all the failures related to a hardware component \((F_1)\), retrieve the dependability model to which it belongs \((dm_1)\)
• **Step (2):** Retrieve the hardware component to which is attached the dependability model \((bus\_bar)\)
• **Step (3):** Retrieve the hardware component which provides the access to the physical medium \((bus\_bar)\) and add to the targeted components set the components which require the access to this physical medium excluding the component found in step 2 (*)\). the Targeted components set = \{system1, system2\}
• **Step (4):** Add to the fault candidates set, all the faults related to the dependability model attached to each element of the targeted component set. fault candidates set = \{f2, f3\}
• **Step (5):** for all the elements in the fault candidates set, compare classification attributes and elect compatible faults.

(*) failure is linked to a service provide by a component to another. “Another” means that a component doesn’t provide a service to itself. In terms of failure/fault compatibility resolution, that’s why reflective propagations are not considered.

c) Software/Software resolution

![Figure 54: software/software compatibility](image)

- **Step (1):** For all the failures related to a software component \((F_1)\), retrieve the dependability model to which it belongs \((dm_1)\)
- **Step (2):** Retrieve the software component to which is attached the dependability model \((soft1)\) and select the outgoing data port related to the expression of this failure (thanks to the relevant AADL binding)
- **Step (3):** Retrieve the software incoming data ports targeted by the previous outgoing port and add to the targeted components set their related software elements. the Targeted components set = \{soft2, soft3\}
- **Step (4):** Add to the fault candidates set, all the faults related to the dependability model attached to each element of the targeted component set. All the fault candidate shall be relative (thanks to the relevant AADL binding) to the previous incoming data ports found. fault candidates set = \{f2, f3\}
- **Step (5):** for all the elements in the fault candidates set, compare classification attributes and elect compatible faults.

4. Behaviour extraction and UPPAAL transformation

At this step, we have a full simplified AADL model where failures are explicitly linked to faults. The last steps of the algorithm are the same as previously seen. From the AADL model the behaviour annex is extracted following the same rules seen in paragraph III.D.2 using the data structure of the next figure. The flow diffusion algorithm is, then, applied in order to destroy the hierarchy and finally only nodes which contain behaviours are kept to generate the UPPAAL configuration file. The little difference comparing with the previous transformation is that transformation rules are not expressed between meta-models. In fact, the
final transformation is processed from the simplified AADL model (which can be seen as an abstraction of the AADL meta-model despite it is not written in MOF format) to the UPPAAL meta-model.

![Diagram](image_url)

**Figure 55 : behaviour annex abstraction**

### V. Dealing with states space explosion

#### A. Observations

The compositional modelling method we chose in the previous section induces a lot of complexity of the final timed-automata generated in UPPAAL. It is an inherent problem with compositional approaches. In fact, if we think about a single global model of the system, simplifications could be made. This induced complexity is added to the initial complexity of the model (FDIR and dependability are not simple notions to model). However in our case we have multiple timed-automata synchronized together. Moreover, comparing to the first transformation approach, decoupling the dependability notions makes us able to enrich the model with complex notions such as dormant faults. This adds complexity too.

Indeed, the results were even worst than expected. The equivalent model of the star trackers on which properties were demonstrated cannot be model checked anymore due to states space explosion. Worst, the complete model built at the end of the study which includes star trackers, reaction wheels, communication buses, and power supply management makes even the UPPAAL simulation tool (not the checker) not to respond properly.

It appears obviously that the proving capabilities depend on the property to be verified. However, what we want to demonstrate are properties, which are of the form: “it is always true/wrong that”. As a consequence, the property form can be considered as an invariant which doesn’t help at simplifying the problematic. So, in order to meet the validation “by proof” objective we have to find either how to build a simpler model - implying that we need to be able to express what is a complex model and what is a simple one - or how to reduce the global model so that it can be checked.

#### B. Complexity criteria

In order to find how to simplify the global model, we have to determine a complexity criterion for a given model. Without considering the simulation problem, this complexity criterion shall necessarily be found with respect to the model checker capability. In other word the question which is asked is: What a complex model, from the model checker point of view, is?

The problem is that we didn’t have any access to the UPPAAL model checker source code. If it was the case, we could have found, by looking at the way the models are analysed, how to express optimized ones. Moreover, the problem of the UPPAAL model checker is that it exploits only the volatile memory of the computer. The states space explodes when the memory is full.
(not other data can be allocated). If we had access to the source code maybe we could have patched the checker in order to add additional swapping to hard disk drive strategies (which enhance the storing capabilities despite computing time grows).

Anyway, the only remaining solution for determining the problems to which the checker cannot face is to build several models which have several design properties and to bench the checker thanks to these models. This solution is very empiric but it is the only solution we found. Anyway, thanks to this method we hoped found a given design property or a combination of properties that are hard to solve from the model checker point of view. Then we hope being able to modify the transformation algorithm to take into account this (ese) property (ies).

In order to find these properties we wrote in the Kermeta language a simple UPPAAL model generator which is able to generate models which are compliant with some design properties. The four axes of research are showed below. The transitions in the models are built thanks to predefined sets of timed, synchronization, or broadcasted synchronization transitions.

Starting from a given population of generated automata, we added complexity in the models by making a balanced combination of growth according to the four axes identified above. However, we couldn’t find very convincing indications, except the fact that the checker is very sensitive to the size of the initial state space. The larger is the initial state space, the longer is the duration of computation. This duration growth is not proportional to the size of the initial state space but indeed, beyond a given threshold the model checker becomes not able to compute the property verification. We cannot prove that it is a clear indication or only a coincidence related to the type of generated models. The last is the most probable because the transitions population is not taken into account (how many broadcast synchronization etc.). Anyway, we observed this tendency.

Based on this observation, we can propose and implement a modification of the transformation algorithm. Considering the general design of the dependability pattern seen before, it appears that the initial state space is very large. In fact, at t₀ the first synchronizations which occur in the system are the faults inhibition. As all the fault propagation models being in their idle state, they are mostly not sensitive to any fault. As a consequence all their linked faults shall be inhibited. As a consequence, to reduce the initial state space size, we modified in the AADL model the initial states of faults to make them passive at t₀. This modification allows having nearly the same proving efficiency than the first designed model. In other words, the AOCS extracted example with the star trackers was successfully proven. However, this little optimization is not sufficient and only pushes away a problem which reappears when implementing a few more devices.

Finally, we are not able to express clearly the complexity of models with respect to a given form of property to be proven and to the model checker capability. The criteria, if they exist, are clearly difficult to identify (especially without information on the checker design) and may involve notions like the dimension of the invariant time interval and the combination of their intersections. As we didn’t have time to investigate further in details, we propose to measure the complexity of a model classically as the number of automata, the number of states/transition and the state/transition dispersion per automata. The complexity of the full model generated at the end of the study is shown below.
C. Fault based model reduction

From the lessons learnt from the previous section, it appears that the UPPAAL generated model shall be reduced. Good news is that the whole model is made of multiple small coupled automata (dependability pattern) instead of less but bigger ones. From this observation, we can say that reducing the number of automata will reduce the complexity of the whole model. There is no huge automaton to simplify (such a simplification is, from a theoretical point of view, very complex). Moreover this observation seems to be consistent with the way we choose to design the model. Fault and failures dynamic are quite simple and fault propagation models can be built in a compositional manner. As a consequence, except isolated complex reconfiguration procedures, there is no big automaton to reduce.

In order to build a reduction algorithm we have to think about the FDIR validation objectives. Validation of the FDIR doesn’t mean compulsory that the whole FDIR strategy in the spacecraft shall be verified thanks to a single property expression. Though it is better, it is not possible according to the model checker capabilities. In fact, a validation process can be efficient even if it is done propagation chain by propagation chain. Among a propagation chain, the FDIR strategy shall be proven to react properly. Moreover, it is generally assumed at dependability analyses level that two (independent) faults never occur at the same time. As a consequence, a simple strategy for reducing the UPPAAL model is to focus on a particular fault (even a set of elementary faults) and only generate the associated propagation chain.

This approach is particularly well adapted to our approach because the transformation algorithm built in the previous section precisely computes the “failures to faults” propagation. In fact, even without analysing the dynamic of fault propagation models it is possible to make a lot of advanced deductions:

- (1): A fault may engender the propagation of the failures linked to the same fault propagation model
- (2): A FDIR component is linked to a given propagation chain if the FDIR monitors at least one fault or failure of this chain.
- (3): A FDIR component is linked to a given propagation chain if it is functionally connected to another FDIR component which is linked to this chain
- (4): A Functional component is linked to a given propagation chain if a FDIR component which is linked to this chain is functionally linked to the component (reconfiguration).

In the following example, two devices (device 1 and device 2) are cold redundant. Focusing on the fault $f_0$, the FDIR shall switch off the failed device and switch on the cold redundant one (squared in bold red are generated model elements).
D. “Step by step” model validation

It has been shown that focussing on a reduced set of faults allows reducing the model size. Anyway, two aspects are still to be addressed. The first is that it might be useless to generate the whole propagation chain because some faults are caught and associated failures recovered at low level (equipment level for instance). The second aspect is that, though the propagation chain is reduced, this reduction may not be sufficient to be verified by the model checker. For the two situations a given chain shall be able to be divided and for the second, chain divisions shall be validated step by step.

The main idea of the chain division is to prove some dependability/FDIR properties on a small part of the propagation chain. Then, the computations allowing to establish this prove lead the system to a given arrival state from which the validation of the next piece of the chain could begin. Dividing the propagation chain means that the propagation of failures is stopped at a chosen point in the model. Those stopped failure are called targeted failures. In other words three classes of components can be distinguished:

- The components which can suffer from the propagation of the fault within the scope of the considered piece of the chain
- The components which can suffer from the propagation of the fault out of the scope of the considered piece of the chain
- The components which cannot suffer from the propagation of the fault.

If we look in time domain, the second category of components cannot be subject to the propagation until the targeted failures are fired. Even if the propagation is instantaneous, there is an ordered sequence of propagation steps. Anyway, until the targeted failure is fired, these types of components remain in a safe state. As a consequence, we introduce the notion of stable state of the system:

“The system is said to be in a stable state, if at a given instant t:

- It is proved that all the components members of the propagation chain piece are in a configuration such that the only fireable transitions are failure propagations event
- Each component, not members of the propagation chain piece, is in a single and predictable \(^2\) safe state”

\(^2\)”Single” means that at the instant t the component can be in one and only one state. “predictable” means that this state can be identified
Anyway, the expression of this rule is valid only considering the way fault propagations are managed. Dividing the chain in pieces raises the problem of propagation loops. A propagation loop will cause the chain to be undividable if not well managed. However our design avoids this problem. In fact, the faults and failures propagations are events based. This means that on the occurrence of the fault an appearance event is fired. After, even if the fault becomes dormant or hidden, the fault is considered as a present fault until the effective disappearance event occurred. As a consequence, even if there is a loop in the model, the presence event will not be fired twice because the fault has already been declared as present. This will break the propagation loop. Based on this loop avoidance property, we can say that if the property of “stable states” is verified at each step of division then the propagation chain can be divided in pieces verifiable independently. However, the problem of predictability of safe states is a problem because it requires analysing the system’s behaviour. However, “safe” components could be extracted from the model to. So, local properties about their dynamic can be proved.

However, this method won’t be able to prove all the properties. In fact it depends on the ability to determine the instant at each division step. In fact, we will consider in most of the case the earliest in case of propagation and the latest in case of recovery. The property will be demonstrated if the recovery (the summation of all the latest) is always faster than the propagation (the summation of all the earliest). Anyway, though it seems possible to express the instant at each step, it may be impossible to prove that in the others parts the arrival states are always single and predictable. Moreover, this approach consider a very pessimistic propagation time because it considers the summation of all the earliest. It is not proven that this summation corresponds to a realistic case.

In the next sections an example of step by step validation will be shown thanks to the reaction wheels example. The following UML activity diagram summarizes the described method.

**Figure 63 : step by step validation method**

### VI. Experiments

At this step we have finished to build the modelling technique and we have gathered in libraries a set of commonly used components linked to both dependability and FDIR aspects. Moreover we are able to deal with the states space explosion thanks to a reduction algorithm added at the end of the transformation chain.

**Figure 64 : full transformation chain**

This section is dedicated to some experiments we made with respect to the modelling method and the proving strategy defined in the previous sections. We do not show all the AADL models here but we focus, in term of fault propagations and FDIR recovery strategy, on some examples which are relevant for the notion explained previously (transient fault, step by step validation, environment etc.). For each example, the corresponding propagation chain is depicted, a reducing strategy is chosen (with associated statistics) and some properties are demonstrated hanks to the results provided by the UPPAAL model checker.
A. Star trackers example – management of transient faults

1. Problem

We do not describe the full propagation case which has been presented in I.C.5 but we will focus in this section on the example of a transient fault occurring at the level of a star tracker electronic device. The idea is to highlight the interest of propagation chain division and to demonstrate some FDIR properties at low level. Even with a reduced number of equipment the faults and failures propagation may enclose many devices. The figure below is an extract of the propagation chain for star trackers.

![Star tracker propagation chain extract](image)

Moreover, even though this is not shown in the figure above, the propagation of the “divergent attitude estimation” (F6) is linked to the reaction wheels sub-systems because the attitude estimation is a measurement input for the wheels control laws. But, considering only this part the propagation chain (and considering that there are three star trackers SST1, 2&3), the verification by proof of the FDIR behaviour cannot be done without facing the states space explosion problem.

2. Reduction

Anyway, if we look in more details at the problem of a transient fault, simplifications can be done. The first, one is the “fault based” reduction (V.C). In our example we focus on the transient internal fault of the star tracker 1 FPGA. So, failures propagation linked to other faults can be removed by the reduction algorithm. This means, the reduction algorithm doesn’t consider the bus A and B and the start trackers 2 and 3. Anyway the problem of the divergent attitude estimation failure propagation remains because the model checker isn’t able to deal with generated models which include the reaction wheels.

In order to stop the propagation at the level of this problematic failure we use a property linked to the transient nature of the fault. As it has been said, transient faults correspond to sporadic events, which can have the same effect as the permanent faults but are more easily recoverable. In most cases a simple reset of the device is sufficient. Besides, in the implemented FDIR strategy this reset signal is fired at functional equipment level after the confirmation of the faults. As a consequence, if the FDIR strategy is well built, the confirmation time and recovery action (including the time need for the tracker to restart) shall be...
shorter than the time needed for the attitude estimation to diverge. In other words, in a case of a transient fault the estimation shall never diverge. If it doesn’t diverge, the related failure is never present so is never propagated. So we can break the propagation chain at this level and validate the FDIR strategy. The reduction algorithm gives the following simplification:

\[
\text{FPGA} \quad \text{Acq.} \quad \text{Bus Coupler} \quad \text{Bus Controller}
\]

Figure 66: star tracker simplified propagation chain

Moreover, the generated statistics shows that the characteristics of the model are conserved. The distribution of state/transition per automaton remains nearly the same. Only the number of automata is reduced (which was the objective).

Figure 67: star tracker example - model reduction 1

Figure 68: star tracker example - model reduction 2

3. Validation

Finally, we were able to express some properties which can be proven by the model checker. For instance, we were able to validate that:
“It is always true that in the presence of a single transient fault on the star tracker FPGA, then the attitude estimation never diverges”

Or

“It is always true that in the presence of a single transient fault on the star tracker FPGA, then the reset signal send by the FDIR makes it disappeared”

In addition we were able to tune some monitoring properties at FDIR level. In fact, most of the monitoring mechanisms in the spacecraft are not compulsory very reactive. In most of the cases there is a periodic mechanism which monitors the presence of a data. This period has a strong impact on the reconfiguration capability. For instance, if the fault occurs just before the monitoring cycle begins then the detection will be very reactive but if the fault occurs just after, the detection ability won’t be available until the next cycle. In order to model this property we use the following monitoring pattern.

Based on this pattern we try to validate some properties of fault detection considering that the fault is injected at $t_0 = 0$ ms. we try to check the following property:

“It is always true that in the presence of a single transient fault on the star tracker FPGA injected at $t0 = 0$, the fault is detected at worst at $t = 200$”

But in a first time it didn’t work. In fact, we made an error in the expression of the property which illustrates the dynamic of faults. The model of the acquisition function implements an initialization time. In other words, at power up, the tracker enters first in an initialisation phase which takes some time. However, during this period, it is considered, that the initialization mode doesn’t process to the acquisition of sky images, so doesn’t need any computing power. This means that though the fault has been injected at $T_0 = 0$ ms and is present in the system, this fault, from the acquisition function point of view, remains dormant until the acquisition phase starts. The property could not be validated because the detection is shifted by the initialization delay. By re-writing the property we could finally validate the detection capability:

“It is always true that in the presence of a single transient fault on the star tracker FPGA at injected $t0 = 0$, the fault is detected at worst at 200 units of time after the star tracker has entered its acquisition mode”

**B. Reaction wheels example – step by step validation**

1. **Problem**

The example of reaction wheels management is interesting for several reasons. It provides a deep propagation chain and introduces propagation cycles. From a general point of view four reaction wheels are used in a configuration such that any three
of them are not coplanar. There are three active wheels and a cold redundant one switched off. The three active wheels are in charge of the satellite rotation among a given axis (one wheel per axis).

Reaction wheels are driven by command laws managed in the onboard computer by the guidance sub-function. This sub-function computes the torque commands to be applied on the wheels thanks to two types of measurements. The first is coming from the “estimation” sub-function and is the attitude quaternion. As it has been said the quaternion can express the rotation axis. The platform position (in radians) with respect to the three active wheels related axes can be deduced from the quaternion. The second measurement is the wheel speed (in radians per second) coming from each wheel.

In the presence of a fault on a wheel, the wheel electronic is in charge of the monitoring (equipment level). The presence detection event is confirmed during a short period, and if confirmed the confirmation event is sent to the AOCs management function which is in charge of the global reconfiguration: the failed wheel is switched off and the cold redundant one is switched on. Three wheels are needed to maintain the attitude. Losing two wheels will lead to enter the safe mode.

For the purpose of this example we focus on an internal fault occurring at the level of a wheel electronic and leading the speed measurement to be lost. Without any FDIR mechanism, in term of propagation, this makes the set point computed for the wheel command law becoming erroneous. At high level, the attitude will become wrong according to the failed wheel related axis after a given amount of time. For modelling, we took the hypothesis that, though all the set points (one per wheel) are computed at “guidance” level, the corruption of a measurement coming from one axis will only engender the corruption of the set point for this axis.

2. Reduction

In this example, a permanent fault is considered. As a consequence the whole propagation chain shall be considered because the fault cannot be recovered early and a full reconfiguration is needed. In term of reduction, we start from an internal fault of the wheel 1 and compute the propagation until the failure linked to the divergence of the attitude is reached. Along the propagation path, taking into account the previous hypothesis, we stop from propagating the failure targeting the wheels 2, 3 and 4. The figures above give the reduced propagation chain and the associated statistics.
3. **Validation**

Though the model is reduced, the propagation is still too large to be verified by the model checker in one single step. As a consequence, we have to iterate the division method in order to split the chain in several pieces on which we can prove independently both propagation and FDIR properties. In this section we apply the method seen at V.D thanks to the following scenario.

At time $t_0$, all the active wheels are powered up. Then they enter in an initialization time until the instant $t_1$. At $t_1$, they enter in a nominal mode in which they start acquiring the set points provided by the "guidance" sub function. At $t_2$ ($t_2 > t_1$) a fault is injected. This fault propagates through the bus coupler and controller according to a time interval $[t_3, t_4]$. Within $[t_3, t_4]$ the set point provided by the guidance function starts diverging. At $t_5$ the real attitude starts diverging until $t_6$ where the satellite’s attitude is not maintained properly. On the FDIR side, the detection is made thanks to a cyclic monitoring. Then the fault is confirmed in time and after the confirmation, the reconfiguration is required.

In this specific case the FDIR strategy shall be validated if the reconfiguration is well performed before the spacecraft attitude becomes wrong. The means that the longest duration needed for recovering the system shall always be less than the shortest time needed for the initial fault to propagate and lead the attitude to become erroneous. As a consequence for the purpose of this example we separate the propagation and FDIR problems.

**a) Propagation validation**

In order to validate the propagation chain, we will split it in two parts. The first part links the initial fault $f_0$ to the failure at the output of the bus controller $F_{51}$ and the second part links the fault $f_{61}$ to the failure $F_7$ and to the fault $f_{11}$. The method iterations steps are depicted below.

**Step 1: model generation**
The model from the $f_0$ to $F51$ is generated. The initial states of the automata are summarized below, all the models are not represented but the principle is the same considering all the automata linked to each component.

<table>
<thead>
<tr>
<th>Component</th>
<th>Automaton</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wheel 1</td>
<td>Fm0</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>Fm1</td>
<td>Off</td>
</tr>
<tr>
<td>Bus coupler</td>
<td>Fm2</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>Fm3</td>
<td>Off</td>
</tr>
<tr>
<td>Bus controller</td>
<td>Fm5</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>Fm8</td>
<td>Off</td>
</tr>
</tbody>
</table>

**Step 2: find the stable state**

Concerning the propagation chain, from the generated model ($f_0$ to $F51$) the following two properties are proven:

“It is always true that in the presence of a single permanent fault on the wheel electronic injected $t_2$, the failure $F51$ can be propagated at time $t$ and not at time $t-1$”

Note: the instant $t$ found is the shortest time needed to propagate the failure $F51$.

“It is always true that if the failure $F51$ is propagated the system have reached the following states space”

<table>
<thead>
<tr>
<th>Component</th>
<th>Automaton</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wheel 1</td>
<td>Fm0</td>
<td>Failed</td>
</tr>
<tr>
<td></td>
<td>Fm1</td>
<td>On</td>
</tr>
<tr>
<td>Bus coupler</td>
<td>Fm2</td>
<td>Fault propagated</td>
</tr>
<tr>
<td></td>
<td>Fm3</td>
<td>On</td>
</tr>
<tr>
<td>Bus controller</td>
<td>Fm5</td>
<td>Fault propagated</td>
</tr>
<tr>
<td></td>
<td>Fm8</td>
<td>On</td>
</tr>
</tbody>
</table>

Moreover, in order to predict the states in which are the other components (not generated) of the propagation chain, the model from $f61$ to $F7$ is generated. From this model it is proven that:

“It is always true that at the instant $t$ the system have reached the following states space:”

<table>
<thead>
<tr>
<th>Component</th>
<th>Automaton</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guidance function</td>
<td>Fm6</td>
<td>On</td>
</tr>
<tr>
<td>Attitude management function</td>
<td>Fm7</td>
<td>On</td>
</tr>
</tbody>
</table>

**Step 3: Iterate on the full chain**

At the end of the step 2 we found the fastest way to propagate the failure $F51$ and we were able to predict the state in which the other components of the chain are. Now we can loop back to a second iteration of the method considering that the clock is fixed, when starting, at $t$ and that the initial states are those presented below.

<table>
<thead>
<tr>
<th>Component</th>
<th>Automaton</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wheel 1</td>
<td>Fm0</td>
<td>Failed</td>
</tr>
<tr>
<td></td>
<td>Fm1</td>
<td>On</td>
</tr>
<tr>
<td>Bus coupler</td>
<td>Fm2</td>
<td>Fault propagated</td>
</tr>
<tr>
<td></td>
<td>Fm3</td>
<td>On</td>
</tr>
<tr>
<td>Bus controller</td>
<td>Fm5</td>
<td>Fault propagated</td>
</tr>
<tr>
<td></td>
<td>Fm8</td>
<td>On</td>
</tr>
<tr>
<td>Guidance function</td>
<td>Fm6</td>
<td>On</td>
</tr>
<tr>
<td>Attitude management function</td>
<td>Fm7</td>
<td>On</td>
</tr>
</tbody>
</table>

The shortest amount of time, $t_{final needed}$ for the whole propagation is found thanks to the following property: “It is always true the failure $F7$ can be propagated at time $t_{final}$ and not at time $t_{final-1}”.

**b) FDIR validation**

The FDIR validation is simpler. We only need the components related to the monitoring and those related to the FDIR directly. In the model the monitoring component is attached to the fault $f11$. As a consequence we generate the model from $f0$ to $F1$ and the associated FDIR strategy. Then from this model we try to find $T_{finalFDIR}$ such that:
“It is always true that in the presence of a single permanent fault on the wheel electronic injected t2, the end of the reconfiguration sequence is reached for all \( t > T_{\text{finalFDIR}} \) and \( T_{\text{finalFDIR}} < t_{\text{final}} \).

Note: “the end of the reconfiguration sequence” means that the wheel 1 has been powered off and the wheel number four has been powered up.

C. Power supply management example – environment interaction

1. Problem

a) Power distribution management

The aim of this example is to highlight how faults and failures can propagate according to some external (environment) constraints. The example chosen is based on the power supply management of the Venus Express spacecraft. The power distribution management is summarized in the following figure:

![Diagram of power management subsystem]

Figure 73: power management subsystem

When the sun is visible, solar arrays provide power to the spacecraft. A solar array is decomposed into panels and each panel is made of photovoltaic cells sections. In the example we consider that each array is made of two panels, each panel including two sections.

The Array power regulators are in fact power converters which are preceded by an adaptive resistor. This resistance is adapted thanks to a Maximum Power Point Regulator (MPPT). In fact the power delivered from a photovoltaic cell depends on the amount of light received. The principle of the MPPT is to impose (by modifying the input resistance) a voltage at the input of the converter and then to measure the delivered power. After a given amount of time a slightly greater voltage is imposed and the power measured again. If the power is greater, the voltage is increased otherwise it is decreased. For dependability purpose the MPPT is in fact a 2 over 3 voting structure where each component measures the power (current and voltage) and proposes a voltage to apply. The voter elects the majority.

Then the current flows down to the batteries to charge them. There are three batteries coupled to dedicated Battery Charge and Discharge Regulators (BCDR) which are in fact energy converters in charge of regulating the incoming and outgoing current with respect to the battery charge and discharge profile. Generally the solar array cannot provide enough power for the spacecraft if it is in a mode in which all the onboard components ask for power. The power peaks are compensated by the batteries. The batteries provide also the power to the spacecraft when the sun is not visible (eclipse).
These two power sources (solar array and batteries) supply a regulated 28 volts primary bus which is split into dedicated dual cold redundant power lines protected by Latch Current Limiters (LCL) or fuses. The means that additional power converters (28V to 5V for electronics boards for instance) shall be found at equipment level. Moreover, the distributed power is also regulated thanks to the Error Amplifier (MEA). Presenting the same structure as the MPPT the MEA is dedicated to balance the power consumed and provided. In fact, if there is more production than consumption the resistances at the input of the solar array regulator are increased in order to dissipate power.

**b) Fault case**

The example on which we focus here concerns a fault that may occur in a section of a solar panel. For a satellite, this kind of fault can be easily monitored because in the presence of the sun the expected power is predictable. This is not the case for an exploration rover for instance. The fault in a section may have the same effect as the light coming in cloudy atmospheric conditions. For the purpose of the example we consider that such faults cannot be monitored even if we deal with satellite. Let’s consider the following situation:

![Figure 74: power supply fault case](image)

- At \( t = t_0 \) a fault on a panel section making this section unable to provide power appears.
- At \( t = t_1 \), the spacecraft enters an eclipse. The solar panel doesn’t provide power anymore, the entire power distribution is ensured by the batteries
- At \( t = t_2 \), the spacecraft temperature is decreasing, and some equipment shall be heat. As a consequence some thermostors are woken up
- At \( t = t_3 \), the spacecraft goes out of the eclipse.

**c) Problematic**

The problem while considering a fault occurring on the solar panel section is that the dynamic of the fault is linked to the presence or not of the sun. Typically, if the sun is visible when the fault occurs, it will be propagated instantaneously otherwise, in the situation of an eclipse, the fault remains dormant as long as the sun is not visible. Moreover, a solar panel shall deliver a service to the battery: “the solar panel provides, in sunny condition, power to the battery”. As a consequence, in the sense of the service delivery, if the section fails, the “charging” service is not well delivered to the battery: The fault propagates from the section to the battery.

In addition, when the propagation reaches the battery the question of dormant and active faults is raised again. For instance, let’s admit that there are (from Venus Express) 8 solar panel sections each providing an equivalent amount of power. If a fault appears on a given section, the power delivered is reduced by =12%. Though the service is degraded, roughly, if the spacecraft stays 12% longer in sunny conditions, the failure at battery level doesn’t have any effect. Furthermore, from the battery, the failure is propagated only if the power expected from the battery cannot be delivered. In other words, the propagation continues only if there is an insufficient state of charge. However, the state of charge depends on the balance of power. How many power producers are active (and how much power do they produce?)? How many consumers are active (and how much power do they consume?). For instance, if the battery is not well charged but if there is no consumer, then the initial propagated fault, at battery level, remains dormant and will never be seen.
For modelling these aspects, we use the environment interface of the dependability pattern we in section IV.C. We propose three environment models. This is not an exhaustive list in the sense of the models used only cover the selected test cases. Anyway the notion of environment model can be extended to more complex notions (mission phases, orbital insertion etc.).

- Night and day model which defines the visibility, from the solar array point of view, of the sun
- Power producer model, which “counts” how many producers are active and their types (low power producer or high power producer).
- Power consumer model, which “counts” how many consumers are active and their types (low power consumer or high power consumer).

The first model is related to solar arrays fault propagation model in order to activate the present faults while the two others are connected to the producers/consumers and to the battery fault propagation models. From these models, the level of charge of the battery becomes easier to master: The speed of charge or discharge of the battery is determined by the number of producers and consumers. Moreover, the presence of propagated fault (or an internal fault) may modify the charge/discharge capability. For the purpose of the example we start, at a given moment, the thermistors (which are high power consumers) according to the following scenario.

![Diagram of theoretical fault propagation](image)

**Figure 75: Theoretical fault propagation**

Concerning the FDIR strategy, a monitoring is implemented at the level of the “insufficient power threshold”. If this event is detected, the payload and the active thermistors are switched off in order to preserve the remaining charge level. For the purpose of the validation we will use the following scenario.

![Diagram of environment and fault injection](image)

**Figure 76: environment and fault injection**
2. Reduction

The aspects of reduction are not really a problem in this example. In fact, we chose to generate the sub-model from the fault related to the solar panel section to the failure related to the total loss of battery power. An extract of the propagation chain is shown below:

3. Validation

The extracted model is fully verifiable in one single step. Its complexity is quite low. Based on the previous scenario and considering that the producers and consumers models drive the speed at which the battery is discharging, we were able to verify how reactive the FDIR strategy shall be in order to prevent the battery from being completely discharged. Considering the maximum discharge speed it also helps at finding at which instant the alarm linked to the insufficient state of charge shall be raised.
VII. Conclusion

At the end of the internship, from observations about the Fault Detection, Isolation and Recovery (FDIR) mechanisms we could define a set of requirements concerning the model based approaches and languages which had to be used. These requirements helped us at finding that not all needed modelling and validation characteristics could be provided by one single language (and associated tools). This led us to select a modelling language (AADL) and a validation language (UPPAAL) and study the unidirectional transformation from the former to the latter.

We elaborate a first modelling approach in order to validate the transformation possibilities. In other words, we validated the transformation techniques which consist in expressing transformations rules over meta-models, but also investigated the capability to model the FDIR concepts. This first approach well represents how the hierarchical, temporal and abstractions aspects could be modelled. Functional and dysfunctional modes are also clearly highlighted. However, the method suffers from usability and scalability issues

Therefore a better structured layered modelling method with improved composability, decoupling and extensibility features has been built. In fact, it answers to several objectives. First, implicit modelling semantic has been introduced in the model. This helped at deducing automatically, according to functional dependencies (hardware/software mapping, data exchanges etc.), how faults and failures could propagate. Then by decoupling dependability concepts of faults, errors and failures, it allowed reducing the scalability problems by introducing a component based approach. Moreover it made possible to model complex faults and failures propagation dynamics. The dependability model built can be plugged to the functional model to form a single consistent model without information redundancy. On this merged model, FDIR elements can be added. In counterpart this approach could offer less semantic aspects concerning the expression of the FDIR hierarchy. To solve this last point, package decomposition has been used.

“By proof” validation was also a hard problem to solve. Well known, states space explosion problem appeared early in the design. Even though we couldn’t find a complete and definitive solution for these aspects we could propose, implement and experiment successfully a set of interesting approaches. First, we extended the transformation algorithm so as to extract fault propagation “branches” based on a reduced set of studied considered faults. Each branch can be validated independently. However, sometimes, even one propagation chain is too complex to be verified. In such case, we built a method based on the decoupling of FDIR and dependability aspects, and on the notion of stable state at a given instant. This method helped us at validating complex chains. Anyway, we cannot say that method works all the time. In fact, it is based on the comparison of propagation time and FDIR reaction time (fastest propagation time > slowest reaction time) and moreover, it imposes that stable states are able to be reached at each division step. However, this method corresponds to worst case validation, so it is very pessimistic. This means that the behaviour is correct if the property is demonstrated but this doesn’t means that the behaviour is not correct otherwise.

At the end, both modelling methods and validation strategies have been benched successfully on representative cases. We studied classical propagations (star trackers), transient faults, propagations loops (reactions wheels), common modes source faults (communication buses) and environment linked faults propagation. In that sense, all the objectives of the internship have been met: a modelling method allowing a component based design, the expression of temporal constraints, fault propagation
and FDIR properties has been built. This modelling method has been evaluated and properties have been demonstrated both on dependability and FDIR aspects.

Anyway, a lot of work remains. First we think that the transformation algorithm shall be enhanced with new propagation deduction capabilities. Then the faults and failures classification used shall be completed and managed transparently by the transformation algorithm. Moreover, we only dealt with simple fault cases and the modelling technique has to be benched against more complex propagations cases. Moreover, the propagation chain division method is not robust enough because it relies on too many propagation hypotheses. Finally, the question of the AADL usage pertinence is raised. We really found this modelling language adapted to the problematic. Anyway one of the interests of the method resides on the assumption that functional models are built in AADL which is clearly not an industrial tendency as SysML seems to be the emergent model based approach.

From a personal point of view, this internship was the occasion for me to approach a lot of notions. In fact, I studied the behaviour of the satellite FDIR functions in several contexts such as feedback control, embedded communication networks or power distribution (energy). Moreover, the study of faults and failures propagation and their recovery procedures helps me at understanding how safety and dependability are managed and especially managed in space domain. In addition, I worked closely to some emerging aspects considering both system design (model based approach) and computer science (meta-model transformation). In the scope of Master in embedded system I found the internship really interesting, challenging and well adapted.
VIII. Bibliography


IX. ANNEX 1: OSATE/Kermeta installation

Step 1: Install OSATE 1.5.4 for Eclipse: osate-topcased-1.5.4-12212007.win32.win32.x86.zip

Step 2: Add the proxy address for internet connection in Eclipse (Windows -> Preferences -> General -> Network Connections)

Step 3: Add the following update site to Eclipse (Help -> Software Updates -> Find and install -> Search For new features to install -> New Remote Site):

- http://download.eclipse.org/eclipse/updates/3.3
- http://download.eclipse.org/modeling/m2t/updates/site.xml
- http://subclipse.tigris.org/update
- http://topcased-mm.gforge.enseeiht.fr/release/update-site3.3/
- http://www.kermeta.org/update/v1.2.0

Step 4: launch the update for all the following sites

<table>
<thead>
<tr>
<th>Sites to include in search:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Eclipse Modeling Framework Technologies (EMFT) Updates</td>
</tr>
<tr>
<td>- Europa Discovery Site</td>
</tr>
<tr>
<td>- Graphical Editing Framework (GEF) Update Site</td>
</tr>
<tr>
<td>- <a href="http://download.eclipse.org/eclipse/updates/3.3">http://download.eclipse.org/eclipse/updates/3.3</a></td>
</tr>
<tr>
<td>- <a href="http://download.eclipse.org/eclipse/updates/3.4">http://download.eclipse.org/eclipse/updates/3.4</a></td>
</tr>
<tr>
<td>- <a href="http://download.eclipse.org/modeling/emf/updates/releases/">http://download.eclipse.org/modeling/emf/updates/releases/</a></td>
</tr>
<tr>
<td>- <a href="http://download.eclipse.org/modeling/m2t/updates/site.xml">http://download.eclipse.org/modeling/m2t/updates/site.xml</a></td>
</tr>
<tr>
<td>- <a href="http://topcased-mm.gforge.enseeiht.fr/release/update-site3.3/">http://topcased-mm.gforge.enseeiht.fr/release/update-site3.3/</a></td>
</tr>
<tr>
<td>- <a href="http://topcased-mm.gforge.enseeiht.fr/release/update-site3.4/">http://topcased-mm.gforge.enseeiht.fr/release/update-site3.4/</a></td>
</tr>
<tr>
<td>- <a href="http://www.kermeta.org/update">http://www.kermeta.org/update</a></td>
</tr>
<tr>
<td>- <a href="http://www.kermeta.org/update/v1.2.0">http://www.kermeta.org/update/v1.2.0</a></td>
</tr>
<tr>
<td>- Model Development Tools (MDT) Updates</td>
</tr>
<tr>
<td>- CSATE Updates</td>
</tr>
<tr>
<td>- The Eclipse Project Updates</td>
</tr>
<tr>
<td>- XML Schema Infoset Model (XSD) Updates</td>
</tr>
</tbody>
</table>

Step 5: install all the packages related to http://www.kermeta.org/update/v1.2.0 (don’t forget to expand all the first nodes and to click on “Select required”)

Step 6: install

Step 7: restart Eclipse

Step 8: Add the following update site to Eclipse (Help -> Software Updates -> Find and install -> Search For new features to install -> New Remote Site):

- http://www.kermeta.org/update

Step 9: launch the update for all the following sites
Annexes

**Dependability and FDIR model based analysis**

### Sites to include in search:

<table>
<thead>
<tr>
<th>Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse Modeling Framework Technologies (EMFT) Updates</td>
</tr>
<tr>
<td>Eclipse Modeling Framework Technologies (EMFT) Updates</td>
</tr>
<tr>
<td>Europa Discovery Site</td>
</tr>
<tr>
<td>Graphical Editing Framework (GEF) Update Site</td>
</tr>
<tr>
<td><a href="http://download.eclipse.org/eclipse/updates/3.4">http://download.eclipse.org/eclipse/updates/3.4</a></td>
</tr>
<tr>
<td><a href="http://download.eclipse.org/modeling/m2r/updates/site.xml">http://download.eclipse.org/modeling/m2r/updates/site.xml</a></td>
</tr>
<tr>
<td><a href="http://subclipse.bgris.org/update">http://subclipse.bgris.org/update</a></td>
</tr>
<tr>
<td><a href="http://topcased-mm.forge.enseeiht.fr/release/update-site3.3/">http://topcased-mm.forge.enseeiht.fr/release/update-site3.3/</a></td>
</tr>
<tr>
<td><a href="http://www.kermeta.org/update/v1.2.0">http://www.kermeta.org/update/v1.2.0</a></td>
</tr>
<tr>
<td><strong>Kermeta update site</strong></td>
</tr>
<tr>
<td>Model Development Tools (NDT) Updates</td>
</tr>
<tr>
<td>OSATE Updates</td>
</tr>
<tr>
<td>The Eclipse Project Updates</td>
</tr>
<tr>
<td>Topcased update site for Eclipse 3.3</td>
</tr>
<tr>
<td>Topcased update site for Eclipse 3.4</td>
</tr>
<tr>
<td>XML Schema InfoSet Model (XSD) Updates</td>
</tr>
</tbody>
</table>

---

**Step 10:** install the package relative to Kermeta IDE (don’t forget to expand all the first nodes and to click on “Select required”)

**Step 11:** install

**Step 12:** restart Eclipse
X. ANNEX 2: AADL reduced meta-model

Figure 80: AADL meta-model simplification
XI. ANNEX 3: AADL behaviour annex reduced meta-model

Figure 81: simplified behaviour annex meta-model used
XII. **ANNEX 4: Star tracker error propagation model**

![Diagram of star tracker error propagation model]

*Figure 82: equipment error propagation pattern*
XIII. Annex 5: error propagation model for double cold redundant buses

Figure 83: error propagation model for double cold redundant buses
XIV. Annex 6: modelling patterns for 1553 bus

Figure 84: controller confirmation

Figure 85: satellite bus reconfiguration decision model

Figure 86: satellite bus reconfiguration action model
## XV. ANNEX 7: dependability, FDIR and environment libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Component name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dependability</strong></td>
<td>sysRootFault</td>
<td>User driveable (root fault) fault model</td>
</tr>
<tr>
<td></td>
<td>sysBasicFault</td>
<td>Chained fault</td>
</tr>
<tr>
<td></td>
<td>sysBasicFailure</td>
<td>Chained failure</td>
</tr>
<tr>
<td></td>
<td>sysTerminalFailure</td>
<td>Failure which cannot propagate</td>
</tr>
<tr>
<td></td>
<td>sysProcFailureModel</td>
<td>Fault propagation model of a computing unit</td>
</tr>
<tr>
<td></td>
<td>sysNumSensorFailureModel</td>
<td>Fault propagation model of a numerical sensor</td>
</tr>
<tr>
<td></td>
<td>sysBusCouplerFailureModel</td>
<td>1553 bus coupler fault propagation model for output data</td>
</tr>
<tr>
<td></td>
<td>sysBusCouplerInputFailureModel</td>
<td>1553 bus coupler fault propagation model for input data</td>
</tr>
<tr>
<td></td>
<td>sysBusControllerOutputFailureModel</td>
<td>1553 bus controller fault propagation model for output data</td>
</tr>
<tr>
<td></td>
<td>sysBusControllerFailureModel</td>
<td>1553 bus controller fault propagation model for input data</td>
</tr>
<tr>
<td></td>
<td>sysBusFailureModel</td>
<td>1553 bus fault propagation model</td>
</tr>
<tr>
<td></td>
<td>sysDualDivergenceFailureModel</td>
<td>Divergence of a computation made from 2 inputs.</td>
</tr>
<tr>
<td></td>
<td>sysControlLoopTwoInputsFailureModel</td>
<td>Fault propagation model of a control loop with two measurement and one set point.</td>
</tr>
<tr>
<td></td>
<td>sysSingleDivergenceFailureModel</td>
<td>Divergence of a computation made from 1 input</td>
</tr>
<tr>
<td></td>
<td>sysGlobal23Divergence</td>
<td>Divergence of a computation made from 2 inputs over 3 possible</td>
</tr>
<tr>
<td></td>
<td>sysSolarFailureModel</td>
<td>Fault propagation model of a solar array section</td>
</tr>
<tr>
<td></td>
<td>sysPowerTransferFailureModel</td>
<td>Fault propagation model of an energy converter</td>
</tr>
<tr>
<td></td>
<td>sysPowerStorageFailureModel</td>
<td>Fault propagation model of an energy storage device</td>
</tr>
<tr>
<td><strong>FDIR</strong></td>
<td>thDirect</td>
<td>Instantaneous fired event on fault presence</td>
</tr>
<tr>
<td></td>
<td>thActiveConfirmation</td>
<td>Confirmation in time of the detection of a fault (cannot be disabled)</td>
</tr>
<tr>
<td></td>
<td>thConfirmation</td>
<td>Confirmation in time of the detection of a fault</td>
</tr>
<tr>
<td></td>
<td>thTimeConfirmedAction</td>
<td>Confirmation in time of the detection of a fault and action fired if the fault presence is confirmed</td>
</tr>
<tr>
<td></td>
<td>thDelayedAction</td>
<td>Delayed action block</td>
</tr>
<tr>
<td></td>
<td>thBurstConfirmation</td>
<td>Confirmation of multiple fault detections in a short period</td>
</tr>
<tr>
<td></td>
<td>thReconfigureCold23</td>
<td>Reconfiguration of 2 over 3 cold redundant devices</td>
</tr>
<tr>
<td></td>
<td>thReconfigureCold34</td>
<td>Reconfiguration of 3 over 4 cold redundant devices</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>sysSunPowerSupply</td>
<td>Night/Day model</td>
</tr>
<tr>
<td></td>
<td>sysDirectEnergyProductionModel</td>
<td>Estimation of the produced energy onboard.</td>
</tr>
<tr>
<td></td>
<td>sysEnergyConsumptionModel</td>
<td>Estimation of the consumed energy onboard.</td>
</tr>
</tbody>
</table>
XVI. ANNEX 8: Properties sets

- sdfPropTime: Time applies to (data);
- sdfPropTypeMainFailure: enumeration (enInternal, enData, enResource) applies to (system, event port);
- sdfPropTypeInternalFailure: enumeration (enBug) applies to (system, event port);
- sdfPropTypeDirection: enumeration (enPresence, enAbsence) applies to (system, event port);
- sdfPropTypeDataFailure: enumeration (enBlocked, enSynthax, enContent, enAbsent, enSpurious, enLate, enEarly, enFrequency) applies to (system, event port);
- sdfPropTypeResourceKind: enumeration (enMicroP, enMemory, enBus, enPowerSupply) applies to (system, event port);
- sdfPropTypeMicroPFailure: enumeration (enComputing) applies to (system, event port);
- sdfPropTypeMemoryFailure: enumeration (enReading, enWriting) applies to (system, event port);
- sdfPropTypeBusFailure: enumeration (enTransporting) applies to (system, event port);
- sdfPropTypepowerSupplyFailure: enumeration (enSoc, enLost) applies to (system, event port);
- sdfPropSensitiveStates: list of aadlstring applies to (system);
- sdfPropFailureModelBinding: list of reference(system) applies to (thread, processor, memory, bus, device, process, system);
- sdfPropUserOwned: aadlboolean applies to (event port);
- sdfUserSpy: aadlboolean applies to (system);
- sdfStopPropagation: aadlboolean applies to (system);
- sdfInitialOf: aadlstring applies to (system, thread);
- sdfInitialClock: aadlinteger applies to (system);
- sdfPropFailureModelOwner: aadlboolean applies to (system);
- sdfFaultBinding: list of reference(system) applies to (data port, connections);
- sdfFailureBinding: list of reference(system) applies to (data port, connections);
- sdfFaultImplementation: aadlboolean applies to (system);
- sdfFailureModelImplementation: aadlboolean applies to (system);
- sdfFailureImplementation: aadlboolean applies to (system);
XVII. ANNEX 9: Quaternion

Positioning (position and attitude) a spacecraft can be represented as a succession of translations from the original position, and a succession of rotations on the successive translations. From a mathematical point of view, the position of the spacecraft can be determined from an original position \((X_0, Y_0, Z_0)\) on which is applied (multiplied) a set of translations \((X, Y, Z)\).

Using matrix representation, the position is determined as follows:

\[
\begin{bmatrix}
1 & 0 & 0 & X_0 \\
0 & 1 & 0 & Y_0 \\
0 & 0 & 1 & Z_0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & X_1 \\
0 & 1 & 0 & Y_1 \\
0 & 0 & 1 & Z_1 \\
0 & 0 & 0 & 1
\end{bmatrix}
\times\ldots\times
\begin{bmatrix}
1 & 0 & 0 & X_k \\
0 & 1 & 0 & Y_k \\
0 & 0 & 1 & Z_k \\
0 & 0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & X_0 + \sum_{i=1}^{k} X_i \\
0 & 1 & 0 & Y_0 + \sum_{i=1}^{k} Y_i \\
0 & 0 & 1 & Z_0 + \sum_{i=1}^{k} Z_i \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

In the same way, the attitude can be characterized by the rotation angle \(\alpha\) with respect to a given axis. The attitude is, as a consequence, the multiplication of a given positioning matrix seen before and the three rotation matrixes possible (one per axis).

\[
R_x(\alpha) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\alpha) & \sin(\alpha) & 0 \\
0 & -\sin(\alpha) & \cos(\alpha) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
R_y(\alpha) = \begin{bmatrix}
\cos(\alpha) & 0 & \sin(\alpha) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\alpha) & 0 & \cos(\alpha) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
R_z(\alpha) = \begin{bmatrix}
\cos(\alpha) & \sin(\alpha) & 0 & 0 \\
-\sin(\alpha) & \cos(\alpha) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Additionally, multiplying these three rotation matrixes patterns give the general expression of a rotation among the \(X, Y, Z\) axes.

\[
R_{xyz}(\alpha, \beta, \gamma) = \begin{bmatrix}
\cos(Z) \cos(Y) & \sin(Z) \cos(X) - \cos(Z) \sin(Y) \sin(X) & \sin(Z) \sin(X) + \cos(Z) \sin(Y) \cos(X) & 0 \\
-\sin(Z) \cos(Y) & \cos(Z) \cos(X) + \sin(Z) \sin(Y) \sin(X) & \cos(Z) \sin(Y) \cos(X) - \sin(Z) \sin(Y) \sin(X) & 0 \\
-\sin(X) & -\cos(Y) \sin(X) & \cos(Y) \cos(X) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

But, the problem with such matrixes is that they are really heavy-weighted to compute and even to transport on a communication bus. It has not to be forgotten that, there is no huge computing power onboard. Due to the energy consumption constraints and to the radiative (aggressive) environment only simple technologies shall be used. As an example, transporting such a matrix using simple precision floating points (considering a classical 32-bits compiler) is equivalent to transport:

\[\text{FloatSize} \times 3 \times 3 = 32 \times 3 \times 3 = 288 \text{bits} = 36 \text{bytes}\]

This approximation doesn’t consider the headers size of a message, the time and size needed for protocols establishment and considers that the fourth column and the fourth line are constant and doesn’t have to be transport.

Considering that attitude determination is used to activate actuators according to command laws (reaction wheels or gyroscopes), the real time constraints are very strict and hard to manage. In order not to oversize communication buses performances and storing ability, this size has to be reduced. This is done using a mathematical tool called quaternion. The main idea is that a quaternion is a single vector of size four which is of the following form (Where \(A, B, C, D\) are real numbers and \(i, j, k\) pure imaginaries such that their product gives the following array)

\[
Q = A \cdot 1 + B \cdot i + C \cdot j + D \cdot k
\]

\[
\begin{array}{ccc}
1 & i & j & k \\
1 & i & j & k \\
-1 & k & -j & 1 \\
-j & -k & 1 & -j \\
k & j & -i & -1
\end{array}
\]
One of the interests of using quaternion is that they are able to express rotations (and any rotations can be expressed as a quaternion). In fact, a mathematical relation expresses the rotation (in 3D space) of an angle $\alpha$ about a vector $\vec{U}$ as the following quaternion.

$$ Q = (\cos \frac{\alpha}{2}, \vec{U} \sin \frac{\alpha}{2}) $$

This relation gives that a quaternion $Q = (q_0, q_1, q_2, q_3)$ represents the rotation such that:

$$ q_0 = \cos \frac{\alpha}{2}, \quad \vec{U} = \frac{1}{\sqrt{q_1^2 + q_2^2 + q_3^2}} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}, \quad \sin \frac{-\alpha}{2} = \sqrt{q_1^2 + q_2^2 + q_3^2} $$

Finally, the entire information that is given by a rotation matrix can be stored in a single vector of four simple precision floating points values which reduces the storing requirement to 16 bytes. Then, if needed, the transformation of a quaternion in a matrix is immediate (and the inverse too) and is well known by programmers (Cf. Bibliography [1])
## XVIII. ANNEX 10: AADL Syntax

### Software component

<table>
<thead>
<tr>
<th>Data</th>
<th>Data are used to strongly type data flows. They can be shared between processes and threads over communication buses</th>
</tr>
</thead>
<tbody>
<tr>
<td>subprogram</td>
<td>Subprograms can be considered as pieces of source code that are executed by a thread or another subprogram</td>
</tr>
<tr>
<td>thread</td>
<td>Threads are light weight processes that represent a program execution flow member of a single addressing space</td>
</tr>
<tr>
<td>threadgroup</td>
<td>A threadgroup is a structural grouping of threads</td>
</tr>
<tr>
<td>process</td>
<td>A process represents an addressing space hosting compiled source code, stack and heap. In AADL they are composed of threads in order to model the structure of a program.</td>
</tr>
</tbody>
</table>

### Hardware component

| Processor | A processor is the execution platform for software parts. This mean that it is an abstraction of both the pure hardware and the scheduling ability of the platform |
| Memory | Memory components represent the physical location where data are stored. They could express both volatile and non volatile memory. |
| Bus | Bus components represent the physical medium allowing the transport of data or continuous notions such as current etc. For Instance, for communication purposes it could be Ether when modelling Ethernet network. |
| Device | Devices are hardware components which are interfaces between software parts and hardware parts. Basically, sensors and actuators are devices. |

### System component

| System | Systems are composite item which allow gathering in the same entity the hardware components, the software components and the connections between them. It is also important to note that a system can be composed of other systems. |