Managing (requirements) evolutions of High Assurance Systems

Lemoine Michel
ONERA
DPRS/SAE
2 avenue E. Belin
31055 Toulouse CEDEX-F
+33 562 252 645
lemoine@onera.fr

Foisseau Jack
ONERA
DTIM/MIB
2 avenue E. Belin
31055 Toulouse CEDEX-F
+33 562 252 634
foisseau@onera.fr

ABSTRACT
Long lifetime HAS (High Assurance Systems) have to deal with numerous evolutions. Managing their inherent complexity can be achieved with an IS (Information System) which records all the artefacts produced / used during their development.

In this paper we present both abstract models than have been designed for representing all the artefacts, and 2 ways of implementing them. We conclude with some lessons we have learned during this study, and that can be reused in any further HAS development.

Keywords
Evolution, Long Life Time System, Information System

1 CONTEXT
Long lifetime HAS (High Assurance Systems) are numerous in the industrial world. Any FMS (Flight Management System) of any commercial or military aircraft is a right representative of HAS because it is highly critical in some different phases of a flight. FMS are designed, tested, and then installed in an operational mode even with the first aircraft prototype in the production line. One of the main FMS characteristics is their long lifetime: 20 years, and very often more! Numerous evolutions appear, either during their development, or during their lifetime.

Aeronautic industry is faced with 3 kinds of evolutions: i) those that are a direct consequence of the arrival of a more reliable and cheaper technology; ii) those that are requested by end-users; iii) those that are applied by the HAS development team itself.

2 CURRENT PRACTICES
In aeronautic and space technology, specific standards [1, 2] are provided for the development of embedded HAS. [1] is mainly devoted to artefacts that must be produced, whereas [2] is more concerned with processes and project management in the context of spatial HAS. Because any embedded HAS must react to its environment, its development is very often also seen as a system development as suggested by the EIA-632 [3].

In space and aeronautic industry, the development of any HAS requires usually the development of 4 products:

- The HAS as developed by the development team – we will call it “generic-product”
- The delivered HAS to the customer, which is a specialisation of the above HAS – we will call it “delivered-product”
- The support system which has helped developing the HAS – we will call it “to-do-product”
- The maintenance system which helps managing the HAS during its lifetime – we will call it “support-product”

The first 2 artefacts are recognised as end-products by the [3], whereas the last 2 are called enabling-products. It must be noticed that the delivered-product is a bit different from the generic-one. Indeed the development team delivers an end-product that does not include variants, specific tests, etc. On the other hand any end-product is specific of a dedicated architecture, including both hardware and software.

Regarding the Airbus A320 family, there exists one, and only one, generic FMS, and dozens of delivered-products.

This practice works for many years. Unfortunately any time an evolution is requested, either for fixing a bug, or for including a new function, or for upgrading some existing components, neither the support-product, nor the to-do-product gives any real help. Each evolution is considered as a modification request. The team in charge of the HAS maintenance analyses it, and makes, when feasible, some proposal to perform the corresponding evolution. But the final decision, accepting or rejecting an evolution, belongs to the customer team. Thus rejection is more often due to economical reasons than scientific ones. The process of tackling evolution is mainly manual: it has not been really considered by standards.

3 PROPOSAL FOR AN IMPROVED PROCESS SUPPORTING EVOLUTION
Our study is built on top of 2 remarks: i) current HAS developments have not been computerised enough, and ii) current processes do not explicitly consider evolution
impact. Consequently our suggestion for a better management of HAS evolution consisted of:

- Recording in a DB (Data Base) all the artefacts that are produced during any HAS development, and its lifetime.
- Designing the DB as an IS whose abstract model takes into account evolution, and makes easier impact analysis.

Our suggestion was not really new since industrial tools such as DOORS [4] allow to consider the description of artefacts, their management, and, the impact analysis of evolution using some \textit{ad hoc} language. Nevertheless our suggestion is: i) more abstract since it does suggest building abstract models; ii) more global since it encompasses the capability of improving the development and support process.

As we wanted to capture all the information that are mandatory to control any HAS development, and its numerous artefacts, it was necessary to use a very accurate notation allowing us to take into account both their static and dynamic nature. This is the main reason we have chosen UML [5]. Other reasons are: i) the ability to give a right semantics to abstract models, even UML is semi formal; ii), the availability of many UML industrial tools, enabling some static and semantic checks.

\textbf{Global Model}

Our IS is based on abstract models representing all the artefacts, i.e. documents and process, produced / used during any HAS development.

In order to be abstract enough, we have considered a generic standard to which we have added evolution aspects. Thus we have designed 7+1 abstract models, right representatives of any HAS and its development.

The 7 models refer mainly to:

- Requirements
- Process
- Physical architecture
- Functional architecture
- Interfaces
- Validation & Verification
- Evolution

The 8th model is the overall model that encompasses all the other ones, i.e. that shows the dependency relationships between all the other packages.

In the following we will focus on 2 examples of accurate abstract models.

\textbf{Requirements package}

In Figure 1, we present the Requirements package in which much information is referenced:

- Requirement
- Problem
- Decision
- Characteristic
- Strategy
- Function
- Hypothesis

All the other information linked to Requirement belongs to other packages. The package name is shown under the class name.

The meaning of all the classes manipulated in the Requirement package is more explicitly given in [6].

- Arcs between classes represent static links, i.e. links that hold for a while. For instance, to any (instance of) Requirement is allocated to a (instance of) Function (which is described in the Logical Architecture package).

Arcs allow also representing some constraints between the instances. Indeed, by means of cardinality, i.e. the numbers of instances at one end that are linked to any instance at the other end of the link we express how many instances are linked.

Regarding the is allocated to association between Requirements and Function we may state that there is a least a function corresponding to any requirement. With such notation we may describe information that are useful for semantics checking. By example, \textit{when creating any HAS we must check that to any Requirements corresponds at least a Function.}

It must be noticed that for presentation reasons we have not presented the cardinality.

- A Category is specialised into 3 subcategories: Functional, Non Functional and Constraints. This is the means to represent a hierarchy from a static point of view.

It is clear that UML is an extremely powerful and simple notation that can be easily taught to anybody. Thus it is easy for end-users, as soon as they are confident enough in the notation, to validate or invalidate models as those presented in Figures 1, and 2.

\textbf{The Evolution package}

In Figure 2 is presented the Evolution package that explicitly gives our semantics to Evolution. It contains mainly four classes:

- Evolution Request, which represents any Evolution.
- Evolution Decision, which records what is the taken decision and why it has been taken.
- Customer Evolution, which is a specialisation of Evolution Request.
- Provider Evolution, which is a specialisation of Evolution Request.

In the Evolution package many links do exist between an evolution and other classes. We have favoured representing the consequence, i.e. impacts of an evolution to all the
The Evolution package allows explicitly tackling the Evolution concept. Any evolution may have impacts onto other classes. In practice we have introduced impacts as numbers following an arc label. For instance in Figure 1, any Requirement \textit{constraints(1)} a Characteristic. In Figure 2, the arc labelled \textit{impacts(2,3)} between Evolution Decision and Product represents 2 kinds of impact.

We have identified 3 kinds of impact:

- (1) denotes an impact linked to a Customer Evolution.
- (2) is associated to a Component change. The component can be either software or hardware.
- (3) is associated to fixing a bug. Indeed, fixing a bug is surely an evolution, but it comes from a different evolution, which is linked to our definition of Evolution in no way.

In Figure 2, the main point is the semantics we have attached to \textit{impacts(…)} associations, thanks to the UML notation. Of course this semantics is a direct translation of industrial practices. When any HAS maintenance team is faced to proceed an evolution, it has to consider all the artefacts that have been produced (when they exist, there are most often paper), and to follow manually links between information items that could be impacted. This very heavy work can benefit from an automation of the evolution package.

4 IMPLEMENTATION OF ABSTRACT MODELS

Having designed all the 7+1 models, a validation phase was done by the industrial team we worked with. They invalidated our first design because the Configuration Management package was missing. After designing it and having introduced it in our set of abstract models, a common agreement was reached. As mentioned above, even software engineers not familiar with UML notation...
can easily understand them.

The main question was then: how to implement the abstract set of models.

We have chosen not to reuse classical tools for managing Requirements Evolution. Indeed it must be noticed that current tools supporting Requirements Management are not satisfactory, from an abstract model point of view. They offer a predefined Requirements Model as suggested by standards which is not open enough to support evolution.

Instead we have chosen the Data Base approach according to 2 complementary implementations.

- The 1st one is more fast prototyping. Using two specific academic tools, TELOS [8] and PROLISP [9], we have been able to translate the abstract models into a set of empty HTML pages. This translation was easy since TELOS does support the Relation concept. These empty pages were then filled in with the right information coming from real documents.
- The 2nd one is more industrial. Using DOORS we have implemented the abstract models as DOORS modules. Nevertheless DOORS is far from offering a pure relational model. It was necessary to consider instances, and not their abstract counterparts. Consequently an important overhead appeared. But, on the other hand, DOORS offers an ad hoc language, namely DXL, which facilitates a lot both navigation and impact analysis of any evolution.

Another way of implementing our abstract models is briefly introduced in the next section. It consists of a set of validation rules that are derived from the abstract models.

Another way of implementing our abstract models is briefly introduced in the next section. It consists of a set of validation rules that are derived from the abstract models.

---

### Figure 2: the Evolution package

5 INDUSTRIAL APPLICATION

The application used for this study is a subset of a navigation system for a large launcher. We are considering the industrial experiments that have been conducted in the following.

**Navigation through the deliverables**

The 1st main application after completion of the HTML pages, equivalent to a DB of distributed deliverables, was the ability, for the customer team, to navigate through all the stored information.

Because the implemented models are a direct instantiation of the abstract models many hyperlinks have been made available! Thus taking advantage of existing browsers, the customer team is able to navigate through all the stored information, in a very different way from what they are used to with the current deliverables they receive from the development team.

Of course HTML pages are far from being a classical DB. It was necessary to add some built-in procedures in order to get some specific navigation, including many atomic steps for getting the right information.

The experiment conducted with the 1st prototype was fruitful. It has allowed demonstrating that:

- The abstract models are accurate.
- The translation of the existing deliverables into HTML pages is feasible and allows taking advantage of existing browsers for a better navigation through the stored information.

A direct consequence of such a fast prototype is that the customer team has been convinced it can communicate in a much better way with the development team. The follow up of the documentation through a shared DB between the 2 teams (the end-users and the developers) allows both
understanding the difficulty the development team is faced with, and, on the other hand, to better follow, and manage, the development itself.

Impact Analysis
The main reason for such a study was to suggest a potential automation of the process of impact analysis whenever an evolution is suggested.

Having designed and developed the above abstract models we have studied how we can automate the analysis of evolution impacts. As written before, the navigation capability offered by the fast prototype allows to identifying what are the information impacted by an evolution.

Indeed it is sufficient to follow the links between items to know what item is impacted! This is implicitly offered by DOORS.

But the impact analysis is much more complicated than a simple link follow up [7]. In order to decide whether there is an impact it is necessary to consider both the recorded design Decision and the Justification (as they are represented in Figure 2). Knowing both the Decision and the Justification associated to a given Requirement allows considering whether there is an effective impact or not, and more accurately if alternative solutions do exist, or why some previous ones have been rejected.

From a practical point of view, we have shown that automating the impact analysis process is dynamic process that needs a complete recording of information acquired during the development process itself.

Unfortunately the current industrial practices do not record design decisions. Thus the impact analysis process is not an automatic one. We have only automated the interaction between the navigation part, i.e. what information is impacted by an evolution request, and the end-user or the developer.

Validation rules
Unfortunately there exist much more developed HAS than to be developed ones. Because from an industrial point of view it is not worthwhile considering the reengineering of existing HAS according to the above methodology we suggested, a crucial question was about its impact on the current industrial practices.

The right answer we have suggested is to translate the abstract models into recommendation rules that are a set of Validation and Verification rules to be applied to the deliverables received by the end-user team. Indeed the abstract models establishes Validation and Verification rules because a semantics is giving through both mandatory links between classes, and their cardinality (not presented in Figures 1 and 2) between instances of objects. This is another advantage of using semi formal notation such as UML: constraints that are designed and represented in abstract models can then be translated by recommendation rules, i.e. V&V rules, to be applied on produced artefacts. Thus we have got a set of checking lists, translating some constraints supported by the abstract models.

It must be noticed that these recommendation rules apply a priori during the HAS development.

6 CONCLUSION
From an industrial point of view, evolution is the main keyword of any long lifetime HAS. With this study we have been able to demonstrate that the management of evolution is feasible, if and only if, the HAS developed is totally recorded, from its first beginning.

The information must be stored in an IS that that can be queried to know the impact of any evolution, before a decision is taken. Moreover, if we are able to enrich the IS with economical information, it should be possible to analysis the evolution impact from a cost point of view.

BIBLIOGRAPHY