Software Engineering Concepts for KBS Design and Testing for Reliability

F. Battini
IEEE Member
via L. Banfi 25, 20040 Carnate (MI), Italy, Tel: +39.2. 25075342
e-mail: battini%laben@icil64.cilea.it

Abstract
The quality of an expert system is usually measured on the goodness of unexpected advice. Although adequacy is involved rather than correctness, however a minimum of correctness should be reached. Hence, as effectiveness of a KBS implies hard predictability on its results, satisfactory testing can be assured just in an operational-like phase of the system lifecycle. Such assumption is not far away by similar assumptions made by some software reliability approaches, like the Cleanroom Approach. To generalize such parallelisms between knowledge and software engineering concepts, this article explains how to provide knowledge engineering with a complete lifecycle strategy.

Keywords
Euclidean Model, Design Principles, Cleanroom Approach

1 INTRODUCTION

KBS are designed for use in non obvious situations where only an expert can infer a sensible solution, whose a priori description is not easy to describe through an a priori analysis and shall be considered as unexpected (i.e. not following a defined and detailed model).

The quality of a KBS is then measured either on the goodness of unexpected advices, if you deal with expert systems, or on the effectiveness of the control actions facing unexpected situations, not explicitly forecasted by the knowledge design.

KBS validation shall start from matching results produced by a fixed set of test cases framing expected results in some simpler situations, whose representiveness, however, can be hardly demonstrated with reference to operations to be faced by the system.

Thus, the concept of absolute correctness should be replaced by the concept of system adequacy: what you can assume as correct by lack of evidence of the contrary, even for results you are not able to predict "a priori". That standpoint dramatically impacts on KBS validation and reliability assessment, unless you fix the following points:

1. Although adequacy is more involved in the KBS reliability rather than correctness, nevertheless a full correctness should be reached at least in simpler situations explicitly forecasted by the knowledge design (Hollnagel, 1991).
2. As a consequence, an operational-like testing is the best suitable way for verifying the KBS adequacy.
3. The goal of KBS reliability should be take into account that KBS do have unpredictable data results (i.e. not framed by a fixed algorithm or model), but must be deterministic in the behaviour of control.
4. Validation and testing are achievable at the end of a development activity, but must be clearly addressed in the earlier design activity.
5. Verification, validation and testing methodology must be framed in a developing strategy referring as most as possible to software engineering principles (Partridge,1988) (ANSI/AIAA,1992).

Our aim is to set relationships between knowledge and software engineering in order to show that reliability assessment can be approached for KBSs through a re-statement of the modularity concept to be managed by knowledge design principles.

We start from the point of view of a knowledge engineer who has not to care on validation and testing of inference engine, user interface and so on, because he is supposed either to use a commercial shell or to have already tested the A.I. tools he is dealing with. His goal is to build a test plan and validation procedures for the implemented knowledge base.

1.1 Knowledge Base as a Software Abstraction

Abstraction is the main feature in knowledge based systems (Partridge,1988). In fact, when we design a knowledge base , we actually design a software in a higher degree of abstraction than the traditional software, which is rather a device, an abstract machine.

For instance the software dealing with the inference engine is an "abstract" engine, whose control is driven by the abstract "software" of the knowledge base. Therefore, no wonder if we abstract the software engineering goals such as modifiability, efficiency, fault-tolerance and understandability as follows:

- **Modifiability** implies controlled changes in the knowledge base such that inference is affected only in the deduction line we desire to change.
- **Efficiency** implies the optimal way in which inferences are performed, pruning everything is redundant or unwanted.
- **Fault tolerance** implies meaningful inferences for every set of data you could input, providing some mechanisms of recovery from failure due to inconsistency, incompleteness or lack of data.
- **Understandability** implies a full understanding of the reasoning lines by the end user: a bridge between problem solving strategy and empirical expertise.

Besides, both software codes and knowledge bases do have primary attributes (the properties) directly felt by the user and manageable by technical principles. Hence, software properties as modularity, concision, consistency, completeness, robustness and self-documentation are still properties of a knowledge base summarized by a second level of attributes: the capabilities.

What we are showing is how to verify, validate and test such capabilities through the modularity of the knowledge base, in order to achieve the goals of fault-tolerance, efficiency and modifiability of our abstracted software: the knowledge base.

1.2 Modularity as a Knowledge Property

Unfortunately the traditional statement of modularity, related to the principles of localization and information hiding, cannot be fully applied in knowledge base design, because knowledge items (rules, frames, script or plans and so on) are strongly coupled to each other. In fact, it could be difficult to decompose a knowledge base in order to readily test it, because localization could be a poor concept in the incremental growing of a knowledge base and it could be hard to define modules cohesively strong and loosely coupled. Nevertheless modularity is just what we are looking for: a knowledge property reached through design concepts, as localization and information hiding, that allows to separate something from the general concern and to verify it without changing its behaviour "in the large". The only thing we would be able to do is to manage the strong coupling among the knowledge items, defining different clusters of related knowledge items and preventing from unwanted interferences among them, through a suitable knowledge model and suitable design principles.

2 THE KNOWLEDGE MODEL

Knowledge grows in an incremental way. Hence a knowledge base, mapping larger and larger expertise, could be thought as a succession of nested incrementing domains, rather than a collection of disjoined areas partitioning a formalized knowledge.

Even a better definition and formalization of a single expertise domain might be viewed as an upper layer of more detailed knowledge encapsulating a less defined knowledge formalization.

Moreover, goals belonging to a single domain could be sub-goals for a more general domain (or a better defined domain) embedding the first one. Hence, that nesting gets domains strongly coupled to each other.

Clusters

If we consider knowledge items sequentially fired in inferencing some conclusions, we have a deduction line. Many deduction lines correlated to the same set of conclusions set up a cluster of related knowledge items. Many deduction lines, making up a cluster, may have in common one or more pieces of their inferencing paths. Hence, a cluster is a set of knowledge items with a strong cohesion.

Module definition

First, we define a module as a cluster of related knowledge items belonging to one or more deduction lines, and not necessarily disjoined, inferencing a correlated set of conclusions and mapping an aspect of a single knowledge domain. The fact that a module has to map just one aspect of a knowledge domain reflects the incremental growing of the knowledge, in such a way you are allowed to merge your module in an outer module mapping a broader aspect, or only a better definition of the first one.

The Euclidean model

We define an Euclidean model of the knowledge, in which 2 kinds of clusters could be defined: theorems and corollaries. Theorems are Goal-driven clusters of knowledge items in backward chaining. Corollaries are data-driven clusters in forward chaining. Hence an Euclidean model of knowledge defines opportunistic reasoning toggling between backward and forward chaining. The knowledge model we defined has been called "Euclidean" because it recalls the logical structure of the Euclid geometry, moving from simpler theorems to more and more complicated deduction with the help of theorem results and corollaries.

3 VVT SUPPORTED BY THE KNOWLEDGE MODEL

The Euclidean model supplies a "natural" methodology for Verification, Validation and Testing (VVT) In fact, theorem results referenced by other theorems can be verified and validated adopting two alternative ways:

- Recursively, asking for details of the referenced theorems (that is by a clear-box testing).
- Through modularity, asking just for which hypotheses inference which conclusions (that is by a black-box testing).

Finally, to test knowledge in solving problem, we must check the correct use of corollaries and theorems referred by a piece of inference.

3.1 Validation and Testing

Validation requires an enhanced confirmability of the knowledge base towards both the understandability goal and the achievement of the right level of correctness. Validation needs a computer-environment interaction model that matches the system dynamic behaviour with reference to possible situations. So, a logically consistent module, that is statically correct, has to behave like a black-box that transforms a time succession of "modulate impulses" into an ordinate succession of meaningful responses, in a sort of convolution between present input and former deductions.

The order of the inferred conclusions becomes very important. In an expert system for instance a right advice given at the wrong time could be hardly understood, because the system is not reasoning as an expert may do. In embedded KBS, a right reaction in the wrong moment can trigger an incorrect overall system response. Therefore the behaviour has to obey to causality in concatenating deductions in the right order. That implies that two kinds of checks should be provided by KBS testing.

- A consistency check, to confirm **statically** the correct relationship between input data and expected results.
- A behavioural check, to confirm **dynamically** the correct evolution of the produced results with reference to a defined computer-environment interaction model.

The difference between the usual software engineering and the knowledge engineering is that knowledge base are continously submitted to testing even after release of modules validated on the basis of a defined test plan. In fact, only the operational use is a meaningful testbed for a knowledge base. Therefore it is no point to put emphasis on testing activities before validation (not to be eliminated, anyway), rather than to be aware that the real testing is just entered when the KBS is released.

Although that may seem awfully misleading to traditional software practitioners (see Adron, 1982), actually there exists an advanced software engineering practice that emphasizes testing on the field and get rid of most test for debugging activities: the Cleanroom Approach (Mills,1987).

Testing is then a natural consequence of the validation activity, because validation checks a limited set of computer-environment interaction situations, while testing is performed in solving real-problems impacting the knowledge base across many domains. If the validation is a minimum test of reliability, testing is a full stress-test as well as an evaluation of the goodness of the computer-environment interaction model.

3.2 The Cleanroom Approach

As KBS testing should cope with hard predictability on the expected data even if the general behavior is well fixed, thus full satisfactory testing can be assured just in an operational-like phase of the system life cycle.

Such assumption is not far away from similar assumptions made by some software reliability approaches, like the Cleanroom Approach (Mills, 1990), that emphasizes the operational testing instead of a mere debugging. Furthermore, that approach uses an incremental design to assess reliable software systems, starting from a first kernel up to successive layers of additional features: just what we should follow to increment a knowledge base.

That similarity sets a relationship between the knowledge engineering and the software engineering practice, giving a baseline in the KBSs implementation for reliability measurements, that can be assessed by a succession of incremental releases, supported by a more and more reliable growing core.

3.3 Reliability Assessment

Reliability for software is a product metrics to be assessed on the final software system after a complete integration, when the software is ready to be exercitated as in the operational use (Musa,1987): as we have seen that is the natural approach to be used in knowledge engineering as well. The measure of the reliability in software can be achieved in 2 ways (Goel, 1985):

1. By means of a continuos monitoring of the decreasing defectiveness in a **Software Reliability Growth** condition typical of integration, verification and validation phases, when the software can still be submitted to corrective maintenance.
2. In a final **qualification** phase by means of statistical models applied on direct observation of presence or absence of failures, when the software is stressed by an actual use. Here the software is frozen and cannot be changed except through a new release.

It must be noticed that testing for validation is aimed at assessing the satisfaction to stated requirements of the produced software, whereas testing for reliability must measure the statistical confidence we have on a validated software to be failure-free in any possible situation. For knowledge base engineering, validation still assesses satisfaction of some requirements that are formalized on the form of very few representative cases, whereas the qualification phase should be considered for KBS a Testing activity for reliability assessment.

Software Reliability Growth

In a Software Reliability Growth condition, the trend in the rate of fault detection and correction is observed and extrapolated in order to foresee the time of the next failure occurrence. Therefore level of the reliability shall be assessed with reference to MTTF (Mean Time To Failure) metrics. An equivalent application of the Reliability Growth Models in the knowledge engineering has been suggested by Bastani (1993), but the trouble stands in the fact that Reliability Growth Models deal with many corrections on the software which may be dangerous on a knowledge base affecting the reliability growing, because the required modularity is hard to get in KBS in order to isolate the fault zone.

Statistical Reliability Qualification

During qualification the software is frozen and its behavior can be observed by means of a statistical sampling of independent possible input classes (**sampling model**) defined following an operational profile (Musa,1993): in our case we talk about triggers and stimuli coming from the environments. By means of the sampling model, assessment of the software reliability shall be made on a statistically meaningful base of tests and again observing the failure rate as in the software reliability growth phase.

In the traditional software, however, the number of the possible observation is very close to 0 by definition, as the software as already passed all the validation steps. Therefore the reliability models used in the former phase, which need a good deal of failure data, cannot be effectively applied. Then the reliability assessment is adopting the Parnas approach (Parnas,1990), a very conservative one, where the so-called **0-failure method** and **k-failure method** are used to measure reliability. Those are an extension to the software testing of observation of a binomial distribution on the results of some Bernoulli trials.

The first method is more conservative, because qualification must restart when a failure, no matter how severe, is observed. The second one allows to keep into account a longer test history, even if some no severe failures are still observed. Then that approach seems more appropriate although:

- The actual use for a KBS is hard to be defined by the equivalent of an Operational Profile.
- According to Poore and Mills (1993) reliability assessment by 0-failure or k-failure methods are very expensive in effort, because to assess a MTTF=τ you should need at least a 2*τ time of observation.

If the actual software production, however, follows an incremental development as in the Cleanroom Approach, the evolution in quality of the software along such sequential releases can be take into account. In fact, it has been observed that MTTF from a release to the next ones increases exponentially:

$$MTTF_p = MTTF_0 \, (e^h)^p \qquad \text{for the } p\text{th release.} \tag{1}$$

where $h = 1 / MTTF_0$ and $MTTF_0$ Mean Time To Failure of the starting release.

A suitable Qualification approach is able to consider such software evolution and then to qualify with less tests a software owning long MTTF, provided that the software has been delivered through p releases. That last framework, referring to the Cleanroom Approach, should be adopted for KBS.

4 APPLICATION FIELD

The described knowledge model and the related VVT methodology has been applied already in 2 real applications:

- The definition of a Test Plan for an Autonomous Spacecraft Data Management System (ASDMS) (Marradi, 1992) where the knowledge base to be validated was a set of predefined plans, fired in a real-time traditional software. In such case, the Euclidean Model supplies the required modularity through data-driven clusters (the Autonomy Plans), but strong coupling between the embedded knowledge base and the background of software processes drives to refine the different goals between the static consistency checks and the dynamic behavior checks (where also real-time concerns had been considered). Besides, according to the described methodology, Validation had been performed on a finite set of deterministic test case, and postponing the "testing" to a qualification phase made directly by the user.
- The development and validation of a production rules expert system, advising on signal processing algorithms, giving some timing and signal constraints. The description of the approach used in that field is explained as a study case in (Battini, 1993).

In the next future the described methodology is likely to be applied on a fuzzy controller where a set of rules shall be defined and validated to control the fine positioning of some actuators.

5 DESIGN PRINCIPLES

It is obvious that efficacy on the VVT techniques may be affected by the testability of the designed knowledge base. Therefore for KBS any VVT methodology should be supported by appropriate design principles able to enhance modularity or at least to manage and reduce strong coupling among different knowledge modules.

Roughly speaking, problems arise when you deal with modules and nested sub modules. To manage strong coupling among them, it is possible, in the case of a knowledge base for an expert system, to state three design principles that can be considered as a general paradigm for translation of design principle from software engineering to knowledge engineering.

Statements

Principle A: every sub module must be fired by an outer module only in these items strictly bounded with the final Goal inferenced by the module.
Desired properties: concision. It is a sort of information hiding principle
Principle B: for every choice between a Goal and one of its subgoals the inferential behaviour, related to the subgoal inference, must be invariant. Or better, the succession of the fired knowledge items must be the same with, eventually, the needed pruning according to principle A. Desired properties: completeness, consistency, modularity.
Principle C: subgoals already inferenced can be integrated and completed through knowledge items in forward chaining belonging to the final goal only.

Desired properties: completeness, modularity.
Figure 1 frames the stated principles with properties, capabilities and knowledge engineering goals.

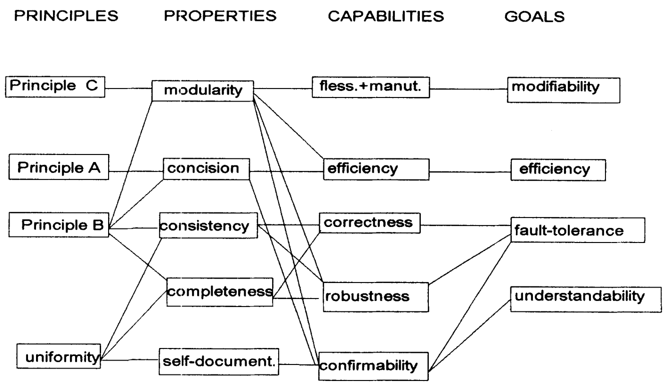

Figure 1 Relationships among design principles, knowledge base properties, capabilities and design goals.

5.2 Applicability of the Principles

To clarify the applicability of the stated design principles, let consider the following example:

- Let N a module, verifying a deduction goal "x" through 2 representation (1 and 2) of the same problem (Figure 2). Either deduction line is chosen according to the initial set S0 of the starting data.
- Nest module N into an outer module M, aimed at verifying final goal "y" through an assessment of the goal "x" inferred by N and representing an intermediate sub-goal of the whole deduction line (Figure 3).
- Suppose that the representation 1 is meaningful for the final goal "y" and that should be used.

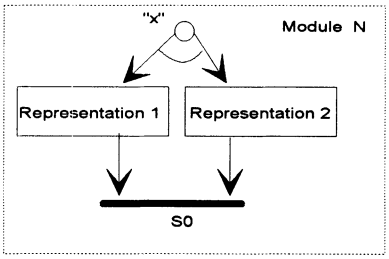

Figure 2 Module by Cluster with 2 representations.

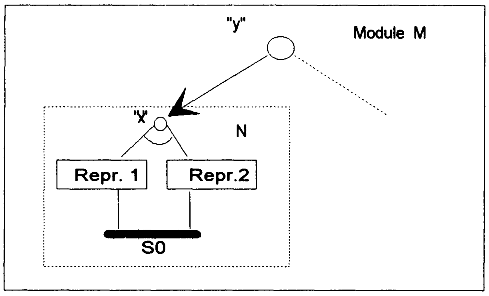

Figure 3 Module M with a nested sub module

Principle A claims that, when the nested (sub-)module N is fired by M for inferencing its final goal "y", just the meaningful deduction line (representation 1) should be scanned and the deduction of "x" should not use any items of representation 2 (Figure 4).

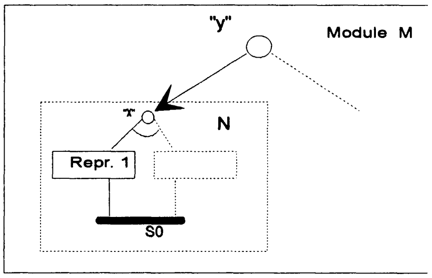

Figure 4 Design Principle A: just the meaningful deduction line (representation 1) should be fired.

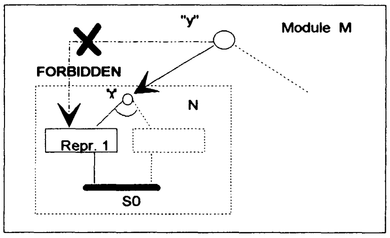

Figure 5 Design Principle B: the communication between modules should be set through sub goals only

Principle B states that no direct links should be set between the final goal "y" and any intermediate items of the representation 1 (Figure 5) to prevent from any interference in the deduction line of the module N. That allows to control coupling between modules, defining the communication mode between them. Besides, principle B states that the inferencing behaviour of the system should not change for any order in the choice between goal "y" and sub-goal "x". Consider the following case:

- First choice: goal "x" through representation 2.
- Second choice: goal "y".

What should happen ? Of course module N should not be fired again, as the sub-goal "x" has been already inferred. Principle C claims that knowledge items transposing the representation 2 into representation 1 should be owned by the outer module M as a set of corollaries and, being data-driven items, should be designed in forward chaining (Figure 6). Principle C, in addition to the module definition, plays the role of localization principle of the "usual" software engineering.

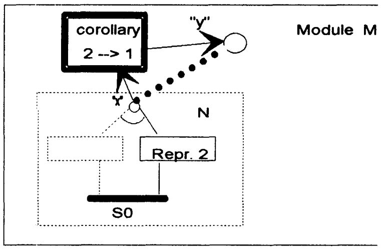

Figure 6 Design Principle C: transformation in the right representation by a forward-chained corollary.

Besides, design principles supply designers with a criterion to define which knowledge items should be chained in backward and which ones in forward.

5.3 Verification

The design principles allow to set the stated VVT methodology, acting in a separate way on every theorem-module, without introducing any trouble during the incremental growing of the knowledge. In fact:

1. cohesion comes from module definition
2. every aspect of a knowledge domain is localised:
   - in a cluster by definition of a module
   - in corollaries belonging to an outer module by principle C
3. coupling is controlled by principle B, because it forbids any link between the final Goal and any deduction line related to the subgoals, defining that the only communication way is through the subgoal.

Moreover, principle C states that every possible link with an intermediate module conclusion must be set through the discontinuity backward/forward chaining.

The result is to get a set of cluster (the theorems), that can be cut in the discontinuity between backward-chained theorems and forward-chained corollaries, holding the property of separability.

Verification is mainly a first consistency assessment in which the order of conclusions has not to be considered. Identifying a knowledge module as a Goal driven cluster of knowledge items embedded in outer clusters, you are allowed to separate a module, because the design principles (especially principle B and C) state how clusters should be coupled.

The discontinuity of the reasoning strategy among cluster links assures the invariance of the behaviour "in the large" for an isolated module. Hence, the module can be separated, applied on a set of initial facts and verified in the inference conclusions through a consistency test.
When a module infers consistent conclusions, according to the knowledge designer purpose, you are supplied with a matrix of linked hypotheses-conclusions that states the consistency, the completeness, the concision of every module.

6 CONCLUSIONS

Reliable KBSs can be assured through effective testing in an environment very close to the operational one. That drives KBS reliability assessment to be approached by a Cleanroom methodology by which successive incremental releases are produced.

As a consequence, the modularity way has to be followed in order to decompose the VVT in simpler activities. Assuming that software engineering goals are still adequate for knowledge based systems (Partridge,1988), we fit them to knowledge engineering through an abstraction.
To get a "cleanroom" product with low defectiveness for each release, a knowledge based system design has to match with:

- a knowledge model
- a computer-environment interaction model, like operational profiles are used for the traditional software

Therefore, VVT should not be thought as a separate activity after knowledge base implementation, but it has to be embedded already in the design phase. Hence the need of design principles, based on the assumed knowledge model, that allow consistency tests and behaviour tests upon more bounded domains of a knowledge base.

7 REFERENCES

Adron S. et alii (1982) Validation, Verification and Testing of Computer Software. *Computing Reviews*, **14**(2), 1982, 159-192.

ANSI/AIAA G-031-1992; *Guide for Life Cycle Development of Knowledge Base Systems with DoD-Std-2167A*.

Bastani,F. and Chen, I.R. (1993) The Reliability of Embedded A.I Systems. *IEEE Expert*, **8**(2), 72-78.

Battini F. (1993) Reliability of KBS: from Knowledge Design to a Verification, Validation and Testing Methodology. Proceedings of the 4th Symposium *Ada in Aerospace*, Brussels;8-11 November 1993.

Goel A.L. (1985) Software Reliability Models. *IEEE Transaction on Software Engineering*, **SE-11**(12), 1409-10.

Hollnagel (1991) The Reliability of Knowledge Based Systems. Proceedings on Workshop *Artificial Intelligence and Knowledge-Based Systems for Space*. ESA-ESTEC, Noordwijk, NL, May 1991 ESA WPP-025, vol.2

Marradi, L. and Battini F. (1992) Verification, Validation and Testing for Autonomous Spacecraft Systems. Proceedings on *Electrical Ground Support Equipment Workshop*, ESA/ESTEC, Noordwijk (NL), ESA-WPP-042

Mills, H.D. and Dyer, M.(1987) Cleanroom Software Engineering. *IEEE Software*.

Mills, H.D. (1990) Cleanroom: An Alternative Software Development Process; in *Aerospace Software Engineering*, (ed. C.Anderson and M.Dorfman), AIAA: Progress in Astronautics and Aeronautics, 1990.

Musa, J.D. Iannino, A. Okumoto, K. (1987) *Software Reliability*. McGraw Hill, N.Y.

Musa, J.D. (1993); Operational Profiles in Software Reliability Engineering. *IEEE Software*, **10**(2), March 1993.

Partridge, D. (1988) *Artificial Intelligence Applications in the Future of Software Engineering*. Ellis Horwood Limited, Chichester.

Parnas, D. and Van Schouwen, A.J., Kwan S.P. (1990). Evaluation of Safety Critical Software, *Communications of the ACM*. **33**(6), 636-648.

Poore, J.H. and Mills, H.D. Mutchler, D. (1993). Planning and Certifying Software System Reliability; *IEEE Software*, **10**(1), 88-99.

7 BIOGRAPHY

Ferdinando Battini was born in Milan, Italy, on October 18, 1960. He received the Master degree in Physics from Genoa University, Italy, in 1987. He was software researcher in the field of signal processing algorithms and A.I. techniques at the Naval System Division of ELSAG - Elettronica S.Giorgio in Genoa. In 1992 he joined LABEN S.p.A in Vimodrone as senior software engineer.

Now, he is supervisor of the software development for the European Photon Imaging Camera (EPIC) experiment for the next XMM spacecraft of the European Space Agency.

Member of the IEEE Computer Society, his reserach interests include Embedded Real-Time Systems, A.I. applications as well as Software Reliability Mesurements.