An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective

Gérard LE LANN
INRIA, BP 105, 78153 Le Chesnay cedex, France
Gerard.Le_Lann@inria.fr

Abstract

The report issued by the Inquiry Board in charge of inspecting the Ariane 5 flight 501 failure concludes that causes of the failure are rooted into poor S/W Engineering practice. From the failure scenario described in the Inquiry Board report, it is possible to infer what, in our view, are the real causes of the 501 failure. We develop arguments to demonstrate that the real causes of the 501 failure are neither S/W specification errors nor S/W design errors. Real causes of the failure are faults in the capture of the overall Ariane 5 application/environment requirements, and faults in the design and the dimensioning of the Ariane 5 on-board computing system. These faults result from not following a rigorous System Engineering approach, such as applying a proof-based System Engineering method. What is proof-based System Engineering for Computing Systems is also presented.

1. Introduction

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure, entailing a loss in the order of 1.9 Billion French Francs ($\sim$ 0.36 Billion US $) and a 1-year delay for the Ariane 5 program. An Inquiry Board was asked to identify the causes of the failure. Conclusions of the Inquiry Board were issued on 19 July 1996, as a public report [3]. The failure analysis given in this paper is based upon the Inquiry Board findings. Our conclusions deviate significantly from the Inquiry Board findings. Essentially, the Inquiry Board concludes that poor S/W Engineering practice is the culprit, whereas we argue the 501 failure results from poor System Engineering practice.

This case study is believed to be particularly useful in illustrating the following facts :
i) current industrial practice vis-à-vis System Engineering for Computing Systems is much more empirical than current S/W Engineering practice,
ii) confusion between System Engineering and S/W Engineering should come to an end,
iii) belief that S/W trustworthiness is the most important challenge faced by the Computing industry need be re-assessed,
iv) the emergence of proof-based System Engineering methods for Computing Systems is one of the great challenges set to our profession for the coming years.

The paper is structured as follows. Section 2 introduces our view of those phases of a computing system lifecycle that pertain to System Engineering. In Section 3, we investigate how some of these phases have been conducted in the case of Ariane 5 (according to the findings published by the Inquiry Board) and we identify what we believe are the real causes of the failure. Conclusions are given in section 4. A detailed analysis of the 501 failure, as well as a critical appraisal of the Inquiry Board report, can be found in [6].

Disclaimer: This analysis is meant to – hopefully – help those partners in charge of and involved in the Ariane 5 program. System engineers cannot be "blamed" for not having applied a proof-based System Engineering method, given that it is only recently that such methods have emerged. This analysis is also meant to – hopefully – explain why it is inappropriate to "blame" S/W engineers.

2. Prolegomena

2.1. What is System Engineering for Computer-Based Systems ?

Following the terminology and the definitions of the IEEE Computer Society, the Engineering of Computer-Based Systems (CBSs) is the engineering discipline which encompasses all the engineering disciplines involved in the lifecycle of CBSs, from the initial expres-

0-8186-7889-5/97 $10.00 © 1997 IEEE

sion of end users/clients requirements up to the maintenance and evolution of deployed CBSs. The Engineering of CBSs encompasses, e.g., Modeling, Human-System Interface Engineering, System Engineering, Electrical Engineering, S/W Engineering.

System Engineering considers CBSs "in the large". A Computing System (CS) is an essential subsystem of a CBS. Hence, System Engineering for CSs considers CSs "in the large" as well, in much the same way Civil Engineers consider bridges or dams "in the large", drawing plans first, then checking they are correct, before starting construction. System Engineering for CSs is concerned with, e.g., external event arrival models, architectures, protocols, algorithms, computational models, failure models, redundant constructs, communications.

We share the view that the essential goal of System Engineering is, for any given project, to produce a global implementation specification of a CS, and to give predictions about the CS future global behavior, along with proofs that predicted global behavior matches the initial expression of end users/clients requirements.

A final global implementation specification of a CS usually is modular, each module being known to be implementable (e.g., over commercial off-the-shelf technology, or via specific H/W or/and S/W developments). However, System Engineering is not concerned with how to implement individual modules. System Engineering serves the purpose of splitting some possibly complex initial problem into a number of simpler and independent sub-problems, in such a way that a solution is known to exist for every sub-problem, i.e. every specification arrived at is implementable as a module (some combination of H/W, S/W and data).

Solving the sub-problems – i.e. conducting a correct implementation of the modules specifications – is the province of Electrical Engineering, S/W Engineering, Device Interface Engineering, Optical Engineering, etc. Implementing a CS module involves specification and design work as well. The crucial observation is that the initial specification of a module cannot be guessed by those in charge of implementing it. Before deciding on how a module is going to be implemented, and then apply relevant Engineering methods (e.g., S/W Engineering), it must be the case that a complete and unambiguous specification of that module is available.

Methods for System Engineering may be more or less "demanding" vis-à-vis formalization or proofs of correctness. For example, specifications may be expressed in human language, in formalized notations, or may result from applying a formal method. In this paper, "specification" is used to refer to any complete and unambiguous statement of a Computer Science problem and/or solution.

There is growing evidence that System Engineering now is the "weakest" (i.e. the least rigorous) of all Engineering disciplines involved with computing systems. Many of the setbacks experienced with projects that involve computing technology are due to poor System Engineering practice. We believe that time has come for the emergence of proof-based System Engineering for Computing Systems, this being one of the major objectives of the IEEE Computer Society's Technical Committee on ECBS [9], as well as one of the goals pursued by INCOSE (Title of the 7th Annual International Symposium in 1997 is "Systems Engineering: A Necessary Science").

2.2. What is proof-based System Engineering for Computing Systems ?

Notation $< N, n >$ is used in the sequel for referring to problems and solutions, N being a set of properties and n being a set of assumptions. Examples of properties are "task atomicity", "timeliness", "dependability". Examples of assumptions are "task execute concurrently", "aperiodic task activations", "network crash failures", "arbitrary processor failures".

SpECS is the acronym used to refer to proof-based System Engineering for Computing Systems. Desired global behavior of a CS, as it should be observed by the CS' users and environment, is specified by the end client/user, via a requirements description (not a specification). Requirements encompass application services as well as the application's environment. Such a description can be viewed as comprising two subsets: one subset $< \Omega, \omega >$, which describes unvalued or partially valued requirements (e.g., type of timeliness requirements is "latest deadlines for application tasks termination", type of overall dependability requirement is "ultra-high availability" [2]), and another separate subset $< \Omega', \omega' >$, which provides valuations of those unvalued variables appearing in $< \Omega, \omega >$ (e.g., actual values of the latest deadlines for every application task, overall unavailability $< 10^{-9}$). Availability of subset $< \Omega', \omega' >$ is not required to start designing a CS.

SpECS covers four important phases of a CS lifecycle. For the sake of conciseness, we will not present the phase concerned with how to cope with changes of user/client initial requirements after a CS has been fielded, or with technological upgrades (existing modules replaced with newer modules). In fact, this phase builds upon the three phases examined below.

- **Capture of initial requirements**: is concerned with the translation of subset $< \Omega, \omega >$ (the application

problem), into a specification of the Computer Science problem "hidden within" $< \Omega, \omega >$, denoted $< \Lambda, \lambda >$, which is also unvalued or only partially valued. Similarly, when available (see further), subset $< \Omega', \omega' >$ is translated into specification $< \Phi, \varphi >$, a valuation of those unvalued variables appearing in $< \Lambda, \lambda >$.

• **System design**: covers all the design stages needed to arrive at a modular (unvalued or partially valued) specification of a CS, the completion of each design stage being conditioned on fulfilling correctness proof obligations. For example, if "mutual exclusion" is a requirement at some design stage, then that design cannot be proven correct unless it includes a mutual exclusion algorithm that matches those assumptions considered at that stage. Similarly, if latest deadlines are to be met by task terminations, one fulfills a timeliness proof obligation by providing: (i) the specification of a scheduling algorithm, (ii) the expression of a computable function B(m, r), which is an upper bound on response times for any task ranked r^{th} in module m's waiting queue, r being shown to be the highest rank reachable by that task, (iii) computable feasibility conditions under which function B holds system-wide.

By the virtue of the uninterrupted chain of proofs that designs are correct, final (unvalued or partially valued) specification of CS – the aggregation of all design specifications, denoted $< D, d >$, provably solves problem $< \Lambda, \lambda >$.

• **System dimensioning**: covers a single stage that has $< \Phi, \varphi >$ as an input. The output is a valuation of $< D, d >$, denoted $< V, v >$. In particular, set $< V, v >$ is a valuation of those implementation variables that appear in $< D, d >$, such as sizes of memory banks, sizes of data structures, processors speeds, databuses throughputs, number of databuses, processors redundancy degrees, total number of processors. Many different subsets $< \Omega', \omega' >$ (hence various subsets $< \Phi, \varphi >$) may be contemplated by a client/user before a decision is made to field or to implement a CS.

Pair $\ll D, d >, < V, v \gg$ is a modular implementation specification of a CS that provably solves problem $\ll \Lambda, \lambda >, < \Phi, \varphi \gg$.

In other words, capture of initial requirements and assumptions yields the expression of a generic problem in Computer Science. System design yields a generic solution of that generic problem.

After this is done, the end user/client is free to dimension his/her generic problem, as many times as desired. For each problem dimensioning, there exists a matching dimensioning of the generic solution (i.e., of the CS). The specification of the matching dimensioned solution is a complete implementation specification of a CS. S/W engineers, Electrical engineers, Optical engineers, Telecommunications engineers, etc. can then be put to work, in order to correctly implement the CS as specified.

A major benefit of applying a SpECS method is design reusability. Whenever a new application problem $< \Omega, \omega >^*$ is being considered, and captured as problem $< \Lambda, \lambda >$, which has been solved in the past with design $< D, d >$, that design comes for free.

It is common practice to assign different pieces of an implementation specification to different teams (subcontractors – competitors sometimes, in-house engineers). Another major benefit of applying a SpECS method is to get rid of three serious problems arising under current practice, which are as follows:

⋆ specifications of some (a few, many) modules are missing, or are ambiguous, or are incomplete; implementors of modules (H/W or/and S/W engineers) proceed by guess-work, which results into errors that, usually, are not caught before the real system is put into operation,

⋆ verification that the set of concatenated modules "behaves correctly" indeed – the "system integration phase" – is a combinatorial problem,

⋆ verification is done via testing, within some limited time budget, which means that verification is almost always incomplete.

Ariane 5 Flight 501 can in fact be seen as a - rather costly - continuation of an incomplete testing procedure.

Under a SpECS approach it is proven beforehand that the reunion of solutions (of modules) is a global solution. Therefore, if every module is correctly implemented, there is no need to check their interactions. The "system integration phase" vanishes (from a theoretical viewpoint) or is vastly simplified (from a practical viewpoint).

Theories and scientific disciplines/techniques that are relevant for SpECS approaches depend on the type of problem under consideration, as well as on the design approach adopted to solve it. Critical applications raise problems of deterministic nature (guarantees for extreme conditions are required). It is then appropriate, or even mandatory, to resort to deterministic design approaches and solutions. Algorithms play an essential role w.r.t. generic solutions and correctness proofs (see [1], [7], [8] for examples). Serializability theory, Scheduling theory, Adversary Arguments, Proofs by contradiction, Worst-Case Analysis, Analytical Calculus, Matrix Calculus in (Max, +) Algebra, are examples of theories and techniques we have found to be useful for solving, with proofs, deterministic application/design problems.

A SpECS method has been developed within INRIA's project REFLECS for addressing those System Engineering issues that arise with Real-time (R), Distributed (D), Fault-tolerant (F) applications and systems. That method (the TRDF method) has been applied to four real problems/cases so far, the most difficult one being a Modular Avionics problem [4]. It is currently being applied to solve a generic R/D/F problem that arises with many critical applications (e.g., Air Traffic Control) and Defense applications.

Analysis of major setbacks, such as project delays, or project cancellations, or operational failures, that occurred over the last ten years, reveals that the dominant cause of such setbacks is neither poor project management nor poor S/W Engineering practice, as is often believed, but lack of applying a SpECS method, which results into faulty capture of initial requirements, or system design faults, or system dimensioning faults, or any combination of the above. This is now illustrated with the 501 failure.

3. Autopsy of the 501 Failure

3.1. The failure scenario

The 501 failure scenario described in the Inquiry Board report is as follows. The launcher started to disintegrate 39 seconds after lift-off, because of an angle of attack of more than 20 degrees, which was caused by full nozzle deflections of the solid boosters and the Vulcain main engine. These deflections were commanded by the On-Board Computer (OBC) S/W, whose input is data transmitted by the active Inertial Reference System (SRI 2). Part of these data did not contain proper flight data, but showed a diagnostic bit pattern of the SRI 2 computer, which was interpreted as regular flight data by the OBC. Diagnostic was issued by the SRI 2 computer as a result of a S/W exception. The OBC could not switch to the backup SRI 1 because that computer had ceased functioning 72 ms earlier, for the same reason as SRI 2.

The SRI S/W exception was raised during a conversion from a 64-bit floating point number F to a 16-bit signed integer number. F had a value greater than what can be represented by a 16-bit signed integer, which caused an Operand Error (data conversion – in Ada code – was not protected, for the reason that a maximum workload target of 80 % had been set for the SRI computer). More precisely, the Operand Error was due to a high value of an alignment function result called BH, Horizontal Bias, related to the horizontal velocity of the launcher. The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4, which results in considerably higher horizontal velocity values.

The Operand Error occurred while running an alignment function, which serves a particular purpose with Ariane 4 but is not required for Ariane 5.

According to the Inquiry Board, causes of the 501 failure are S/W specification and S/W design errors. We now present those System Engineering faults which, in our view, are the real causes of the 501 failure.

3.2. Capture of initial requirements

Under a SpECS approach, stating problem $< \Lambda, \lambda >$ involves establishing a specification of the following (in particular) :

- set of external events (modeling of the CS environment),
- for every external event :
  - ⋆ list of every (possibly one) application-level task that is activated,
  - ⋆ arrival law (periodic, sporadic, aperiodic, arbitrary),
- for every application-level task:
  - ⋆ external variables shared by this task and the CS environment, as well as variables that depend on those external variables (rc_1),
  - ⋆ application-level conditions under which that task can run, be suspended, resumed, aborted (rc_2),
  - ⋆ timeliness constraints (latest termination deadline, bounded termination jitter, etc.).

3.2.1 Fact

Horizontal Velocity is an external variable, shared by the environment and some of the SRI module tasks. Horizontal Velocity is used to compute the value of an integer variable called BH. This value, which is sent to the OBC modules, determines the nozzle deflections (solid boosters, Vulcain main engine).

Fault RC_1 (external variable): The need to split the initial application requirements into two subsets $< \Omega, \omega >$ and $< \Omega', \omega' >$ has not been identified. A fortiori, the need to transform description $< \Omega, \omega >$ into specification $< \Lambda, \lambda >$ has not been identified either. Hence, problem $< \Lambda, \lambda >$ has not been stated. Consequently, external variables, such as Horizontal Velocity, and related variables, such as BH, have not all been listed explicitly (see rc_1 above).

This fault has resulted into system dimensioning fault DIM_1.

3.2.2 Fact

The alignment program was running after lift-off. The "exception condition" (BH overflow) was raised while running this program. It was later acknowledged that there is no need to keep this program running after lift-off in the case of Ariane 5.

Fault RC_2 (application task attribute): Had problem $< \Lambda, \lambda >$ been stated, the alignment task would have been listed, along with its attributes. In particular, the condition "alignment task should not be running after lift-off", would have been explicitly specified (see rc_2 above). Which has not been done.

This fault has resulted into system design fault DES_5 . It was simply impossible for those system engineers in charge of designing/selecting the SRI computer task scheduler, as well as eligibility scheduling rules, to "guess" that the alignment task should have been aborted right after lift-off.

3.2.3 Fact

Continuous correct SRI service is essential to a correct functioning of the launcher. This has not been the case.

Fault RC_3 (dependability property and assumptions): A specification of problem $< \Lambda, \lambda >$ would have stated the following dependability property DEP and related assumptions:

(DEP) for some SRI module failure model, SRI service is correctly and continuously delivered for the entire duration of a flight, with a probability at least equal to 1 - p.

In other words, up to f SRI modules - out of n - "can" fail during a flight. Probability that more than f such failures can occur should be p at most.

Rather, it was implicitly assumed that SRI computers (H/W only, unlike modules) would behave according to "crash-only" semantics. It was also implicitly assumed that 1 SRI module at most would fail. This has resulted into system design faults DES_1, DES_2, DES_3 and DES_4, as well as into system dimensioning fault DIM_2.

3.2.4 Summary

Beyond the fact that, via the reading of the Inquiry Board report, it is possible to diagnose the three faults listed above, one may wonder whether the current Ariane 5 on-board CS is plagued with design/dimensioning faults other than those identified in this paper, some of them being ascribable to RC faults other than RC_1, RC_2, RC_3. Problem is that conclusions and recommendations listed in the Inquiry Board report focus almost exclusively on S/W Engineering issues. It is reasonably obvious that the expertise required to identify RC faults (fault RC_3 in particular) and system design/dimensioning faults is not related to S/W Engineering culture in any respect. Issues raised are Systems Engineering issues.

The current Ariane 5 on-board CS is a solution to some unknown problem $\ll \Lambda, \lambda >^*, < \Phi, \varphi >^* >$. Reverse System Engineering should be applied in order to identify this problem. However, problem $\ll \Lambda, \lambda >, < \Phi, \varphi \gg$ not being stated, it is impossible to compare $\ll \Lambda, \lambda >, < \Phi, \varphi \gg$ and $\ll \Lambda, \lambda >^*, < \Phi, \varphi >^* >$. Portions of problem $\ll \Lambda, \lambda >, < \Phi, \varphi \gg$ – such as property DEP – can be inferred. Consequently, related design/dimensioning faults are identifiable (see further). Such faults can be corrected. However, doing this does not suffice. As long as problem $< \Lambda, \lambda >$ will not be completely and unambiguously specified, it will be impossible to assert that the Ariane 5 on-board CS is correct. Even if flights 502 and 503 turn out to be successful.

Note: no SpECS method was applied either in the case of Ariane 4. It was experimentally "verified" in the course of real launches that potential faults in requirements capture and/or system design/dimensioning/implementation were ironed out (some faults having possibly contributed to a few early failures). Such "experimental" approaches may yield a "confidence level" comparable to that reached with proving, albeit in a much more costly and lengthy manner. Now that SpECS approaches have emerged, why should we keep wasting time and money?

3.3. System design

Under a SpECS approach, proofs are mandatory at every single design stage. For example, for specified failure models, proofs that DEP holds for such failure models should be given, for some feasibility conditions to be established. Let n be the smallest number of SRI modules needed to cope with up to f module failures. Making such "deterministic" assumptions as f-out-of-n (property DEP), or any similar assumption of probabilistic nature, does not make sense unless it can be demonstrated that there are no faults that can cause n identical failures (the "common mode" failure scenario). Two approaches can be followed:

- either prove that requirements capture and system design/dimensioning/implementation faults are avoided; in this case, non-diversified redundancy can be considered,
- or skip such proofs (which has been done); it is then mandatory to consider diversified redundancy, in order to tolerate potential faults, with a sufficiently

"high" probability. The greater the value of f, the higher the probability.

This is an important observation. Many "fault-tolerant" CSs are implemented via non diversified redundancy, despite the fact that (i) their design is based on f-out-of-n assumptions, (ii) no proofs that system design and system dimensioning are correct are given. This is like building on quicksand, given that the activation of any System Engineering fault is guaranteed to lead to a total CS failure (n failed modules), whatever the respective values of f and n.

3.3.1 Fact

Requirement that property DEP should hold was violated. Both SRI modules failed.

Fault DES_1 (the SRIs): No SpECS method was used. Hence, non diversified redundancy is an incorrect solution (see above).

Other Space missions have failed for the same reason (Mars Observer is a recent example).

3.3.2 Fact

Both SRI modules failed, exhibiting failure behavior stronger than "crash-only", namely "omission" and "erroneous non-malicious behavior in the value domain".

Fault DES_2 (the SRIs): The implicit "crash-only" assumption has not been specified. Hence, it could be violated, which happened. Detection-and-recovery - which is the type of fault-tolerance algorithm used - cannot be proven to be a correct solution unless failure models are specified.

For property DEP to hold, it is well known that active redundancy is the only algorithmic construct that can be considered in the presence of unspecified failure models (i.e. stronger than "crash-only" a priori). It is also well known that $n > 2f$ is a necessary and sufficient feasibility condition under such circumstances.

Fault DES_3 (the SRIs and the OBCs): The implicit "crash-only" assumption has not been specified. Hence, rather than "killing themselves", SRI modules issued diagnostic/error messages.

Under a SpECS approach, every event that can be posted to a module must be listed. Similarly, tasks that can be activated over a module must be listed. A {task, event} mapping must be provided. Had this been done, event "flight data" could not be confused with event "exception condition reporting". Consequently, System engineers would have identified the need to provide for (among others) two separate OBC tasks, one in charge of executing the flight program, the other one in charge of handling exception conditions (error messages). Had this been done, "crash-only" behavior would have been correctly implemented vis-à-vis the OBC flight program.

Although this would not have helped avoid the 501 failure, this is a latent fault that may lead to a future flight failure.

3.3.3 Fact

The OBC modules (H/W + S/W + data) in charge of commanding the nozzle deflections interpreted error messages issued by the SRI modules as flight data.

Fault DES_4 (the OBCs): No proof has been established showing that a correct OBC module delivers correct inputs to the OBC task in charge of commanding the nozzle deflections, in the presence of some faulty values issued by the SRI modules.

A correct OBC module can be shown to compute a correct value out of a set of values issued by n SRI modules, f of them at most being faulty, by using an error masking algorithm (e.g. majority voting). A harder problem, known as the "Byzantine Generals" problem, was solved long ago, for the most unrestricted failure model [5]. This work has influenced the design of the US Space Shuttle on-board CS.

In passing, one may observe that this "checking" conducted by the OBC modules would have compensated for the lack of checking (of BH values) by the SRI modules. And the 80 % maximum workload target set for SRI computers would not have been exceeded. This has not been done due to lack of having rigorously looked at the on-board CS "in the large".

3.3.4 Fact

Loss of correct flight data occurred while running the alignment program. With Ariane 5, keeping this program active after lift-off is unnecessary.

Fault DES_5 (the SRIs): Conditions under which tasks can be run, suspended or aborted are to be stated explicitly (see rc_2, section 3.2).

Had this been done, and assuming a correct task scheduler has been designed for the SRI modules (not a S/W Engineering issue), the alignment task would have been deactivated (by the task scheduler) as specified, e.g. right after lift-off if so desired.

3.4. System dimensioning

No rigorous requirements capture having been conducted, and design faults having been made, provably correct dimensionings of some of the variables appearing in the final on-board CS implementation specification were missing (BH and f, in particular). That could

not be detected during the (S/W and/or H/W) implementation phase, unless accidentally. This resulted into an incorrect size of the memory space used for storing variable BH. This resulted also into an empirical dimensioning of the SRI modules group (which should have been based on diversified redundancy).

3.4.1 Fact

The Operand Error was due to an unexpected high value of BH.

Fault DIM_1: Range of values taken by variable BH was assumed to be in the $\{-2^{15}, +2^{15}\}$ interval.

Had a SpECS method been used, fault RC_1 would have been avoided. Hence, the client/user or some representative – e.g., a space/flight engineer – would have been explicitly asked to quantify the ranges of possible values of Horizontal Velocity and BH, for the application under consideration, that is Ariane 5 (not Ariane 4). This would have translated into a correct dimensioning of the memory block used to implement BH.

Similar implicit assumptions may have been made also for other variables or data structures.

Calling an incorrect dimensioning of a memory block a "S/W design error" (see [3]) is as erroneous as calling the selection of too slow a processor a "S/W design error". Again, the dimensioning of external variables is a System Engineering issue, not a S/W Engineering issue.

3.4.2 Fact

Both SRI modules failed. The f-out-of-n (1-out-of-2) implicit assumption was violated.

Fault DIM_2: It was implicitly assumed that f's value would be 1 and that simple redundancy (n = 2) would suffice. No proof that probability 1 - p would be higher than some quantified value, was given.

The intrinsic reliability of a SRI module (H/W + S/W + data), and related coverage factor, can be derived from statistics computed over accumulated experimental data, or via probabilistic modeling. Intrinsic reliability and coverage factor, combined with launch duration, are used to compute a value for f, as well as related probability q that at most f failures "can" occur. Obviously, the smallest correct value of f to be chosen should be such that 1 - q < p holds true.

It is thereafter possible to derive the lower bound of n which, depending on the (non malicious) failure model that should have been considered, is f+1 or 2f+1 (the latter in our case).

It appears that System engineers took H/W only into consideration. This has also led to a violation of the n > 2f lower bound, which applies whenever non malicious failure semantics are not specified.

Although a value of f (resp. n) greater than 1 (resp. 2) would not have helped avoid the 501 failure, this is a latent fault that may lead to a future flight failure.

4. Conclusions

It is now possible to understand what led to the failure of flight 501. If we were to stick to internationally accepted terminology, we should write the following:

Faults have been made while capturing the Ariane 5 application/environment requirements. Faults have also been made in the course of designing and dimensioning the Ariane 5 on-board CS. These faults have resulted into errors at the system modules level. Some of these errors have been activated during launch. Neither algorithmic nor architectural constructs could cope with such errors. Flight 501 failure ensued.

Had all the modules - as specified - been implemented in H/W, flight 501 would have failed the way it did. Had the implementation of every module (in S/W and/or in H/W) been proven correct - w.r.t. the specifications at hands - flight 501 would have failed the way it did. Therefore, causes of the failure belong neither to the "S/W world" nor to the "H/W world". Issues at stake are System Engineering issues.

Our conclusions deviate from those of the Inquiry Board. We have found that these conclusions are unjustly directed at S/W Engineering errors. This is yet another example of a long lasting confusion between S/W Engineering and System Engineering. Despite evidence that S/W engineers should not be blamed for the 501 failure, improving the quality of the S/W design and production process is a reasonable recommendation, as usual. (It is almost always the case that S/W can be improved). However, it is essential to understand that looking at CSs as S/W "things" inevitably leads to disillusions. And to erroneous or inaccurate failure diagnostics. It is regrettable that manifestations (S/W errors) are confused with the real causes (System Engineering faults).

Regrettable but, maybe, not surprising. Expertise required to identify – or, even better, to avoid – System Engineering faults is not at all related to S/W Engineering culture.

A particularly striking illustration of the above is as follows. We have identified – among others – a serious system design fault, which mirrors knowledge that is reasonably widespread in the System Engineering community, namely that any CS based on non diversified redundancy inevitably fails whenever a system design or a system dimensioning fault is activated, whatever

the redundancy degree. Obviously, such knowledge – hence the fault – was ignored by the Inquiry Board. The Board found no weakness w.r.t. the existing redundant on-board CS, which is demonstrated by the diagnosis that "there is considerable redundancy at the equipment level"[3]. Non diversified simple redundancy is the weakest type of redundancy that can be imagined.

It is reasonably clear that neither architectural nor algorithmic issues have been carefully investigated by the Inquiry Board. More testing is recommended by the Board. We argue that System Engineering faults cannot be detected by testing, unless accidentally.

Proof-based System Engineering eases the job of S/W engineers (faults that would result into S/W errors are avoided). Conversely, proof-based S/W Engineering does not compensate for poor System Engineering practice. Similarly, in Civil Engineering, "good" reinforced concrete cannot compensate for lack of having global plans proven correct in the first place.

Whether or not the existing Ariane 5 on-board CS, or any future Ariane 5 on-board CS, is free from design and/or dimensioning faults other than those identified in this paper can only be established by applying a SpECS method.

A fortiori, it is strongly recommended that a SpECS method be applied to other Space programs.

References

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publisher, ISBN 0-201-10715-5 (1987), 370 p.

[2] J. Gray and D. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, pages 39–48, Sept. 1991.

[3] Inquiry Board. ARIANE 5 - Flight 501 Failure. *Inquiry Board report, http://www.inria.fr/actualités-fra.html (July 1996)*, 18 p.

[4] INRIA Project REFLECS. Algorithmique TR/TD/TF ORECA. *4 reports (in French), French DoD contract DRET 94/395*, 1995-1996.

[5] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems, vol. 4, 3*, pages 382–401, July 1982.

[6] G. Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. *INRIA Research Report 3079 (Dec. 1996)*, 26 p.

[7] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publisher, ISBN 1-55860-348-4 (1996), 872 p.

[8] M. Joseph, et al. *Real-Time Systems - Specifications, Verification and Analysis.* M. Joseph Editor, Prentice Hall UK Publisher, ISBN 0-13-455297-0 (1996), 278 p.

[9] S. White, J. Rozenblit, and B. Melhart. Engineering of Computer-Based Systems: Current Status and Technical Activities. *IEEE Computer*, pages 100–101, June 1995.