Proof-Based System Engineering and Embedded Systems

Gérard Le Lann

INRIA, Projet REFLECS, BP 105
F-78153 Le Chesnay Cedex, France
E-mail: Gerard.Le_Lann@inria.fr

Abstract. We introduce basic principles that underlie proof-based system engineering, an engineering discipline aimed at computer-based systems. This discipline serves to avoid system engineering faults. It is based upon fulfilling proof obligations, notably establishing proofs that decisions regarding system design and system dimensioning are correct, before embarking on the implementation or the fielding of a computer-based system. We also introduce a proof-based system engineering method which has been applied to diverse projects involving embedded systems. These projects are presented and lessons learned are reported. An analysis of the Ariane 5 Flight 501 failure serves to illustrate how proof-based system engineering also helps in diagnosing causes of failures.

1 Introduction

Taurus (stock exchange), Relit (stock exchange), AAS (air traffic control), Confirm (on-line hotel and car reservation), Socrate (on-line railways seat reservation), Freedom (manned orbital station), P20 (nuclear power plants), Ariane 5 Flight 501 (satellite launcher). This is a small sample of projects that have been significantly delayed, or have been cancelled, or have entailed costs much higher than anticipated, or have resulted into operational failures, because of problems with "informatics". According to a study conducted over 8,380 computer-based system projects by the Standish Group in 1995, only 16% of these projects were found to be successful, i.e. completed in time, within their initial budget, and having produced systems that - as of 1995 - had not failed.

Industrialized countries are wasting huge amounts of time and money for the simple reason that our community is not very good at designing and building computer-based systems that match clients requirements at decent costs. Among those computer-based systems that function properly, how many are unjustifiably expensive, in terms of development and maintenance? Our community is not very good either at correctly identifying the real causes of project setbacks or operational failures. Too often, blame is inappropriately put on poor project management and/or poor software (S/W) engineering practice.

There is growing evidence that system engineering currently is the weakest (i.e., the least rigorous) of all those engineering disciplines covered by what the IEEE Computer Society refers to as the Engineering of Computer-Based

Systems. This view is not universally shared (yet). In certain circles, the motto is "S/W is all what matters". Interestingly enough, infatuation with S/W is manifest in some of those countries which have failed so far to develop a profit-making hardware (H/W) industry. That S/W is not a dominant cause of project setbacks or system failures is supported by a growing number of studies (see [10], [14], [16] for examples).

Asap, Better, Cheaper (ABC) now is the clients defined rule of the game. Meeting the ABC rule involves dramatic improvements in methods, processes and tools directed at systems engineering in general, at system engineering for computer-based systems in particular. This need has been acknowledged by the US industry and governmental bodies in the early 90's. As a result, the InterNational Council On Systems Engineering (originally called NCOSE) was created.

Let us now consider the body of knowledge that has been accumulated over the last 35 years by the research community in computer science. It is reasonably straightforward to conclude that a large fraction of those system engineering problems faced by the information industry and by the computing industry have solutions readily available. To be more specific, there are many architectural and algorithmic problems that have well documented optimal or correct solutions, notably in the areas of real-time computing, distributed computing, and fault-tolerant computing. These are the areas of relevance for embedded applications and systems. However, only a small subset of these solutions have so far translated into commercial, off-the-shelf (COTS) products or operational computer-based systems.

That time has come for major changes in system engineering (SE) methods, practices and tools directed at computer-based systems is no longer being questioned in professional circles. Which sorts of changes is the issue of interest. We believe - and we will attempt to convince the reader - that such changes should be aimed at introducing correctness proof obligations, as is (resp. is becoming) the case in the integrated circuits (resp. software) industry. We believe that proof-based system engineering for computer-based systems - referred to as proof-based SE in the sequel - is the privileged vehicle to successfully meet the ABC challenge. Many embedded applications being complex and/or critical, meeting correctness proof obligations seems even more appropriate in the case of embedded systems.

Proof-based SE is introduced in section 2. In section 3, we present a proof-based SE method - the TRDF method - and we show how to conduct the various phases of a computer-based system project according to this method. Some real projects addressed with the TRDF method are sketched out and lessons learned are reported in section 4. Section 5 serves to illustrate how proof-based SE, when applied a posteriori, helps in diagnosing the real causes of operational failures. The Ariane 5 Flight 501 failure is the case selected.

2 What is Proof-Based System Engineering

2.1 Introduction

With proof-based SE, one seeks to solve system engineering problems arising with computer-based systems in much the same way engineering problems are solved in such well established disciplines as, e.g., civil engineering or electrical engineering, that is by "exploiting" scientific results. Indeed, before they undertake the construction of a dam or the electrical cabling of a building, engineers draw plans first (design) and check that plans are correct (proofs). Of course, they do not have to re-establish theoretical results, such as theorems in hydrodynamics or the Kirchhoff laws. They apply well established engineering methods which build upon (and "encapsulate") fundamental scientific work. Why would it be different when it comes to computer-based systems?

The essential goal pursued with proof-based SE is as follows: starting from some initial description of an application problem, i.e. a description of end user/client requirements and assumptions (e.g., an invitation-to-tender), to produce a global and implementable specification of a computer-based system (denoted S in the sequel), along with proofs that system design and system dimensioning decisions made to arrive at that specification do satisfy the specification of the computer science problem "hidden" within the application problem.

With proofs, it is possible to ensure that future behavior of S is the desired behavior, before implementation or construction or fielding is undertaken. Proof-based SE aims at avoiding system engineering faults, in contrast with fault-tolerance approaches, such as those based on, e.g., "design diversity".

Any project involves a number of actors, namely a client, a prime contractor, co/sub-contractors. Roles taken on by actors are described wherever appropriate. Note that a client may not be a specific organization or company, but a team of marketing/business experts internal to a system vendor/integrator company that targets a certain market, in which case another team (of engineers) would take on the role of a prime contractor.

2.2 Three Essential Phases in a Project Lifecycle

Notation <Y> (resp. [Z]) is used in the sequel to refer to a specification of a problem Y (resp. a solution Z). Notation <y> (resp. [z]) is used to refer to a specification of a set of unvalued problem-centric (resp. solution-centric) variables. The term "specification" is used to refer to any complete set of unambiguous statements - in some human language, in some formalized notation, in some formal language.

Proof-based SE addresses those three essential phases that come first in a project lifecycle, namely the problem capture phase, the system design phase, and the system dimensioning phase. Proof-based SE also addresses phases concerned with changes that may impact a computer-based system after it has been fielded. For example, phases devoted to handling modifications in client-originated descriptions of application problems, coping with evolutions of COTS

products, taking advantage of advances in state-of-the-art in computer science, and so on, are also covered by proof-based SE. Such phases simply consist in repeating some of the three phases introduced below, and which precede phases covered by other engineering disciplines (e.g., S/W engineering), which serve to implement system engineering decisions. The whole set of notations needed to describe these phases is summarized in figure 1.

$$
\begin{array}{rl}
< A > & \equiv \text{specification of an invariant (i.e., generic) application problem } A. \\
< X > & \equiv \text{specification of the generic computer science problem that} \\
 & \quad \text{matches } < A >. \\
< a > & \equiv \text{specification of the set of variables in } < A > \text{ that are left unvalued.} \\
< x > & \equiv \text{specification of the set of variables in } < X > \text{ that are left unvalued.} \\
[S] & \equiv \text{modular specification of a generic computer-based system } S \text{ such} \\
 & \quad \text{that } [S] \text{ satisfies } < X >. \\
[s] & \equiv \text{specification of the set of variables in } [S] \text{ that are left unvalued.} \\
[oracle.S] & \equiv \text{specification of a dimensioning oracle (obtained from correctness} \\
 & \quad \text{proofs) for generic system } S. \\
V(< x >) & \equiv \text{some assignment of values to variables in } < x >. \\
V([s]) & \equiv \text{an assignment of values to variables in } [s] \text{ that satisfies } V(< x >).
\end{array}
$$

Notations

$$
\{\text{description of an application problem}\} \Rightarrow < A > \left| \begin{array}{l} \Rightarrow < X > \\ \Rightarrow < a > \\ \Rightarrow < x > \end{array} \right.
$$

Capture

$$
< X > \left| \begin{array}{l} \Rightarrow [S] \\ \Rightarrow [s] \\ \Rightarrow [oracle.S] \end{array} \right.
$$

Design

$$
V(< x >) \Rightarrow oracle.S \Rightarrow V([s])
$$

Dimensioning

Fig. 1. Organization of proof-based SE phases

The Problem Capture Phase

This phase has an application problem description as an input. Such a description, provided by a client, usually is expressed in some human language and is incomplete and/or ambiguous. The capture phase is concerned with, (i) the translation of an application problem description into $< A >$, which specifies the generic application problem under consideration and, (ii) the translation of

$< A >$ into $< X >$, a specification of the generic computer science problem that matches $< A >$. A generic problem is an invariant for the entire duration of a computer-based system project.

Specifications $< A >$ and $< X >$ are jointly produced by a client and a prime contractor. A client knows "what he/she wants", i.e. the semantics of the application problem of interest. A prime contractor is in charge of identifying which are the models and properties commonly used in computer science that have semantics that match those of the application problem. Consequently, a specification $< X >$ actually is a pair $\{< m.X >, < p.X >\}$, where m stands for models and p stands for properties.

For example, statement "*despite being read and updated possibly concurrently, files should never be inconsistent*" in $< A >$ would translate as "*serializability property (for operations on files)*" in $< X >$. Or statement "*workstations used by air traffic controllers should either work correctly or stop functioning*" in $< A >$ would translate as "*dependability property is observability = stop failure (for workstations)*" in $< X >$. Or statement "*how application-level service* app *will be invoked is unknown*" in $< A >$ could translate as "*unimodal arbitrary arrival model (for the event type that triggers* app)" in $< X >$ (see section 3.1 for an introduction to models and properties).

Besides making sure that a client fully understands what is implied with every model and property that appears in $< X >$, a prime contractor is in charge of helping a client decide which degree of genericity is appropriate. Inevitably, variables appear in specifications $< A >$ and $< X >$. Notation $< a >$ (resp. $< x >$) is used to refer to a specification of those variables in $< A >$ (resp. $< X >$) that are left unvalued. As for $< X >$ (resp. $< A >$), $< x >$ (resp. $< a >$) is a pair $\{< m.x >, < p.x >\}$ (resp. $\{< m.a >, < p.a >\}$.

The genericity degree of $< A >$ or $< X >$ may vary from 0 ($< a >$ and $< x >$ are empty) to ∞ (every variable in $< X >$ (resp. $< A >$) appears in $< x >$ (resp. $< a >$)).

Of course, any degree of genericity has an associated cost and a payoff. With a "high" degree of genericity, design work is "hard", but is valid for a very large number - say N - of problem quantifications and, therefore, releases of S, which entails possibly significant savings vis-à-vis design, dimensioning, implementation and testing activities. Conversely, with a "low" degree of genericity, design work is "easy", but it must be repeated "many" times in order to deliver these same N releases of S. Hence, design costs are higher. However, the cost of every single (specifically dimensioned) release of S is lower than that of any release resulting from a "highly" generic approach. (See also further, under the system dimensioning phase sub-section, and section 2.3).

Set $\{< A >, < X >, < a >, < x >\}$ is the contract established between a client and a prime contractor. Such a contract is essential for avoiding those misunderstandings or litigations that are commonplace with current system engineering practice. Any extra cost or delay incurred in a project due to changes in $< A >$ or $< a >$ is to be beared by a client. Similarly, any extra cost or delay due to erroneous $< X >$ or $< x >$ (i.e., any mistaken translation of $< A >$ or

$< a >$) is to be beared by a prime contractor. There are methods and tools (e.g., rapid prototyping) that may help a client produce a specification $< A >$ that correctly mirrors "what he/she has in mind". One of the purposes of proof-based SE methods is to help specify $< A >$ and $< a >$ as well as correctly derive a set $\{< X >, < x >\}$ from a given set $\{< A >, < a >\}$.

At present time, such derivations can be performed by experts, namely computer scientists and engineers who are aware of such methods. We foresee the existence of knowledge-based tools that will help automate these derivation processes.

The System Design Phase

This phase entirely is under the responsibility of a prime contractor. A design phase has a specification $< X >$ as an input. It covers all the design stages needed to arrive at $[S]$, a modular specification of a generic solution (a generic computer-based system), the completion of each design stage being conditioned on fulfilling correctness proof obligations. A design phase is conducted by exploiting state-of-the-art in various areas of computer science (e.g., computing system architectures, algorithms, models, properties), in various theories (e.g., serializability, scheduling, game, complexity), as well as by applying appropriate proof techniques, which techniques depend on the types of problems under consideration (examples are given in the sequel).

More precisely, one solves a problem $\{< m.X >, < p.X >\}$ raised at some design stage by going through the following three steps: specification of an architectural and an algorithmic solution designed for some modular decomposition, establishment of proofs of properties and verification that a design correctness proof obligation is satisfied, specification of a dimensioning oracle.

For example, let $\{<$ *distributed processes, some partially synchronous system model* $>, <$ *mutual exclusion* $>\}$ be a problem contemplated at some design stage. One cannot proceed further in the design phase unless, (i) one specifies a modular architecture that supports the distributed processes, (ii) one proves that the required safety property (mutual exclusion) is ensured with some to-be-specified distributed algorithm designed for a computational model "as strong as" - see section 3.1 - the specified partially synchronous model.

Similarly, if a real-time problem such as $\{<$ *distributed task models, multi-modal arbitrary arrival model* $>, <$ *latest deadlines for completing task executions* $>\}$ is to be solved, one goes through that design stage by, (i) specifying a distributed modular architecture that supports the specified tasks, as well as a distributed scheduling algorithm, (ii) proving that the specified timeliness property holds for every task for feasibility conditions "as strong as" the specified arrival model.

The first stage of a design phase results into the breaking of the initial (possibly complex) generic problem $< X >$ into a number of independent generic subproblems. Fulfilling a design correctness proof obligation guarantees that if every subproblem is correctly solved, then the initial problem is correctly solved as well by "concatenating" the individual generic solutions. And so on. Consequently, a design phase has its stages organized as a tree structure (see fig. 2).

By the virtue of the uninterrupted tree of proofs (that every design decision is correct), the union of those specifications that sit at the leaves of a design tree is a modular specification of generic S that provably correctly satisfies $< X >$. This modular specification is denoted $[S]$. If $< X >$ is a correct translation of $< A >$, then, transitively, $< A >$ is provably correctly solved with $[S]$.

Clearly, this approach is based on compositionality principles very similar to those that underlie some formal methods in the S/W engineering field.

Every module of $[S]$ is deemed implementable, or is known (in a provable manner) to be implemented by some procurable product or is handed over to a co/sub-contractor. As will be seen in section 3, another output of a design phase is a specification of a system-wide dimensioning oracle - denoted $[oracle.S]$ - which includes, in particular, a set of constraints called (system-wide) feasibility conditions. Of course, $[oracle.S]$ must be implemented in order to conduct subsequent system dimensioning phases. From a practical viewpoint, $[oracle.S]$ is a specification of a $\{< X >, [S]\}$-dependent component of a more general system dimensioning tool.

The System Dimensioning Phase

The purpose of a dimensioning phase is to find an assignment $V([s])$, i.e. a quantification of system S unvalued variables, such as, e.g., sizes of memory buffers, sizes of data structures, processors speeds, databuses throughputs, number of databuses, processors redundancy degrees, total number of processors. $V([s])$ must satisfy a particular assignment $V(< x >)$, i.e. a particular quantification of the captured problem-centric models and properties. A dimensioning oracle, i.e. an implementation of $[oracle.S]$, is needed to run a dimensioning phase. Such an oracle may be owned by a client or a prime contractor, or both.

One or several dimensioning phases may have to be run until the dimensioning oracle declares that there is a quantified S that solves a proposed quantified problem $\{< X >, V(< x >)\}$. How many phases need be run directly depends on the genericity of $[S]$. Consider for example that $[s]$ is close to empty, for the reason that many (design and dimensioning) decisions were made prior to entering the design phase. This is typically the case whenever it is decided a priori that S must be based on specific COTS or proprietary products. The good news are that a small number of dimensioning phases need be run, given that many system variables are valued a priori. The bad news are that the oracle may find out (rapidly) that the proposed problem quantification is not feasible (e.g., some deadlines are always missed), no matter which $V([s])$ is considered.

Pair $\{[S], V([s])\}$ is a modular implementation specification of a system S that provably solves problem $\{< X >, V(< x >)\}$. Modules of $\{[S], V([s])\}$ are contracts between a prime contractor and those co/sub-contractors in charge of implementing S.

To summarize, a capture phase yields the specification of a generic problem in computer science that matches an invariant application problem. A design phase is the resolution of that generic problem, which yields a specification of a generic solution, exactly like in Mathematics where one demonstrates that some theorems hold - properties "as strong as" those stated in $< p.X >$ - for every

possible set of values taken by some set of variables var, for some axiomatics - models "as strong as" those stated in $< m.X >$. After this is done, a client or a prime contractor is free to quantify the generic problem, as many times as desired. For each feasible quantification, there exists a matching dimensioning of the generic solution (i.e., of S). To pursue the analogy with Mathematics, theorems are applied for various assignments of values to set var.

If deemed implementable, set $\{[S], V([s])\}$ is the borderline between system engineering on the one hand, S/W engineering, electrical engineering and other engineering disciplines on the other hand.

S/W engineering serves the purpose of producing correct executable implementations of given specifications. Where do these specifications come from? Obviously, they result from system engineering work. Hence, S/W engineering necessarily follows system engineering in a project lifecycle. Why is it useless or, at best, marginally productive to apply formal methods in the S/W engineering field without applying proof-based methods in the system engineering field? For the obvious reason that provably correct S/W implementations of specifications that are flawed in the first place can only result into incorrect computer-based systems. For example - as amply demonstrated by many failed projects and/or operational failures - a faulty design decision results into specifying some inadequate algorithmic solution. No matter how correctly that specification ends up being implemented, the resulting system can only fail to meet $< X >$. Specifications handed over to S/W engineers must first be proved correct w.r.t. some client-originated problem.

2.3 Stochastic Versus Deterministic Approaches in Proof-Based System Engineering

System engineering for computer-based systems has been an area of concern for almost half a century. Until recently, with few exceptions, design and/or dimensioning correctness proofs that have made inroads in industrial and business circles are those based on stochastic approaches, namely probabilistic approaches (e.g., analytical modeling, queueing theory) and statistical approaches (e.g., numerical simulation, event-driven simulation) [1].

We believe time has come for changes. We hope that the viewpoints presented below will help to properly put current trends into historical perspective.

The major reason for changes is the clients defined ABC rule. Consider the trends in the embedded applications area. Can we still "play with probabilities" when, on the one hand, complexity and criticality go beyond certain thresholds and, on the other hand, costs must go down, costs including losses incurred because of operational failures? With embedded applications, it is more and more often the case that those computer science problems derived from client originated problems are of deterministic nature. Typically, clients want to be shown that properties $< p.X >$, quantified as per $V(< p.x >)$, always hold under assumptions $< m.X >$, quantified as per $V(< m.x >)$, i.e. for possibly exceptional

[1] Proofs serve the purpose of predicting before constructing. Consequently, we do not consider here such techniques as prototyping or testing.

or worst-case operational conditions. Most often, deterministic approaches are then mandatory.

For example, whenever initial $< X >$ indirectly raises a real-time, or a distribution, or a fault-tolerance issue, it is necessary to resort to such deterministic algorithms and "non probabilistic" proof techniques as those found in, e.g., [1], [5], [7], [15]. This is so for at least one reason: the set that includes all possible runs of a reasonably complex computer-based system cannot be built by, (i) resorting to models based on (independent) random variables, (ii) considering stochastic "adversaries".

Let us imagine we have to solve a distributed hard real-time problem. Any solution must include some distributed synchronization algorithm (e.g., concurrency control, consensus). Therefore, those variables that model individual processor states are related to each other in very specific ways. They cannot be seen as independent random variables. Furthermore, upper bounds on response times must be given for worst-case scenari such as, e.g., highest "load" densities (a deterministic "adversary"). This is radically different from expressing expected values and standard deviations of response times assuming, e.g., Poisson arrivals.

Composite algorithms built out of, e.g., concurrency control algorithms, fault-tolerant algorithms, distributed algorithms for fault-tolerance, real-time scheduling algorithms, on the one hand, reasoning techniques such as adversary arguments (game theory), proof techniques pertaining to mathematical logic, and such calculi as matrix calculus in (max, +) algebra, on the other hand, are needed to establish desired solutions and correctness proofs.

Of course, stochastic approaches have their own merits. Whenever appropriate, they should be followed. Nevertheless, we would like to dispel the following misconceived argument: "Given that "the world" is probabilistic, only stochastic approaches make sense". Firstly, it has never been proved that "the world" is probabilistic. Probabilities and coverage factors are one possible manner of modeling our inability to tell the future (of "the world"). Secondly, the fact that real future operational conditions may from time to time deviate from what has been assumed does not mean that it is impossible to prove properties "deterministically".

More to the point, we see three fundamental differences between deterministic and stochastic approaches.

The first one is related to the capture phase. As will be seen, in any of the classes of models that are considered under a deterministic approach, there is one model that is "extreme", in the sense that it reflects a fully unrestricted "adversary". Examples of such models are the asynchronous computational model, the byzantine failure model, the multimodal arbitrary event arrival model. Picking up these models has a price: one may run into impossibility results or one may end up with "demanding" feasibility conditions (which translates into costly computer-based systems). The beauty of such models is that they are "safe": no real future operational conditions can be worse than what is captured with these models. Therefore, the issue of estimating coverage factors is void with such models. Conversely, even with "extreme" stochastic models, the issue of estimating

coverage factors must be addressed. Picking up models weaker than "extreme" ones involves a risk: operational conditions may be "stronger than" assumed during a capture phase. Again, this is unavoidable, given that we cannot "tell the future". However, it is fallacious to conclude from this (rather trivial) observation that only stochastic approaches make sense. "Deterministic" theorems are established under Euclidian or Riemannian axiomatics, without asking which matches best some given universe. The same is true with properties in computer science. Picking up the appropriate models for $< m.A >$ and $< m.X >$ boils down to making tradeoffs between having coverage factors sufficiently close to 1 and retaining models as weak as acceptable.

The second fundamental difference is related to the design phase. There is no probability or coverage factor involved with deterministic design approaches. Properties "as strong as" those stated in $< p.X >$ either hold or do not hold. For example, mutual exclusion or serializability or atomic broadcast or some timeliness property is either not ensured or it is (via some deterministic algorithm), under specific feasibility conditions. There are no probabilities or "proof coverage factors" or "proof confidence intervals" involved with proofs of properties or feasibility conditions.

Feasibility conditions are analytical expressions. For a given set of models, they define a set of scenari that, with certainty, includes all worst-case scenari that can be deployed by "adversary" $< m.X >$. (This set is exact, i.e., is not a superset, whenever feasibility conditions are necessary and sufficient). There are no probabilities or coverage factors involved with expressing computable lower bounds on a redundancy degree or on a number of modules, or computable upper bounds on response times. Conversely, there is inevitably some approximation involved with a stochastic modeling of those deterministic algorithms (probabilistic ones, in some instances) that constitute the inner mechanisms of computer-based systems.

Let us consider probabilistic approaches first. Even though the modeling of some deterministic arrivals models and algorithms is known to be tractable in queueing theory (e.g., ND/D/1), approximations cannot be avoided whenever queue servers model real processors that are actually synchronized. This raises the issue of estimating the level of error introduced with such approximations. Common practice consists in validating probabilistic analytical models via event-driven simulations. Hence the question: What can be expected from statistical approaches ?

Given that random number generators and limited numbers of input scenari are resorted to under such approaches, it is impossible to ascertain that every possible worst-case scenario has been explored. Coverage factors or confidence intervals depend on the sizes of the actual state spaces. In most real cases, these sizes are huge (see sections 3.4 and 3.5 for numerical examples). Which results into coverage factors or confidence intervals considered realistic or satisfactory when in the [95 % - 99 %] interval. This is problematic with critical applications. When highest accepted unavailability of critical services is set to 10^{-4} (resp. 10^{-7}) by a client (via $V(< p.x >)$) - as is the case with some

airborne/spaceborne (resp. air traffic control) applications - we can no longer rely on stochastic proofs of properties, for the simple reason that levels of confidence reachable with probabilistic analytical modeling or statistical modeling have a distance to 1 which is orders of magnitude greater than 10^{-4} (resp. 10^{-7} a fortiori).

The third fundamental difference is related to the dimensioning phase. In both cases, quantifications of models (via $V(< m.x >)$) and properties (via $V(< p.x >)$) have some related coverage factors. However, in the deterministic case, there are exceptions. For example, quantification coverage factors need not be estimated when considering "extreme" models (e.g., the asynchronous computational model, the byzantine failure model) or such properties as serializability or timeliness. This being granted, it is important to see that, under a deterministic approach, physical dimensionings of computer-based systems have no coverage factors or confidence intervals or probabilities associated to them. Those numerical values that appear in $V([s])$ specify a "safe" physical dimensioning of a computer-based system. A "safe" dimensioning is a by-product of checking the feasibility of some problem-centric quantification $V(< x >)$. Consequently, such dimensionings are certainly sufficient (ideally, they should be necessary and sufficient), contrary to stochastic dimensionings, which are probabilistically or statistically correct.

We do not mean to dismiss the relevance of stochastic approaches. What is getting clearer is the existence of a strong trend in favor of resorting to deterministic approaches more often than has been the case in the past. Interestingly enough, besides embedded applications, such a trend is manifest in application domains that have traditionally been terra incognita for deterministic approaches such as, e.g., telecommunications [8]. In the sequel, we consider deterministic proof-based SE approaches and methods.

2.4 Current Status

As is the case with other engineering disciplines, various proof-based SE methods will emerge in the future. Work on proof-based SE which was started in 1993 [11], [12] has led to the development of a method named TRDF [2]. That name was retained for the reason that those application problems of interest to us raise real-time issues (R), or distribution issues (D), or fault-tolerance issues (F), or any combination of these issues. Embedded applications raise such issues. TRDF currently is the only proof-based SE method (based on a deterministic approach) we are aware of. Consequently, in what follows, we further explore proof-based SE as instantiated by the TRDF method.

[2] Temps Réel, Traitement Distribué, Tolérance aux Fautes (T is a common factor)

3 Proof-Based System Engineering and the TRDF Method

In this section, we provide a detailed description of what is involved with design and dimensioning phases. In order to do this rigorously, detailed notations are necessary. They are provided for the interested readers. However, these notations may be skipped by those who simply want to read this paper as an introduction to proof-based SE.

3.1 The Problem Capture Phase

Most often, properties (services to be provided to end users) and models (assumptions regarding future operational conditions) are not clearly separated in an application problem description. Furthermore, such descriptions inevitably contain many TBD (to-be-defined) quotations. This is felt to be a major cause of those difficulties faced by a prime contractor and, transitively, by co/sub-contractors. It should not - and may not - be so. Too often, clients believe - or are told by prime contractors - that they must make early commitments on specific numerical values for variables appearing in the descriptions of their application problems.

Of course, this results into specific - hence, simpler - design problems. Conversely, this maximizes the likelihood that some of these values are changed by a client after a design phase has started or is completed. As a result, most often, existing design work is invalidated. This does not happen if invariants $< A >$ and $< a >$ are identified.

As explained in section 2, a capture phase involves a number of interactions between a client and a prime contractor. The identification of invariants $\{< A >, < a >\}$, as well as their translation into $\{< X >, < x >\}$, rest on combining knowledge of application-centric semantics and knowledge of those models and properties elaborated in computer science. For example, only a client can tell whether α-particles should be of concern. If such is the case, a prime contractor is responsible for translating this into appropriate failure models.

Let us now give a non exhaustive list of classes of models and properties (each class being non exhaustively described either). Every class has a hierarchical structure. In some cases, partial orders are known (see, e.g., [3], [17]). Of course, this is an area of on-going research. When element Y precedes element Z in the hierarchy or in the partial order considered, one says that Y dominates Z, or that Y is stronger than or equal to Z, which is written $Y \supseteq Z$. Notation $\supseteq$ correctly reflects the fact that Z is included in Y or that $Y = Z$ (see further for examples).

This definition of dominance or strength is consistent with the fact that work involved with a design phase is best described a la game theory, as follows. Set $< m.X >$ defines an "adversary". A designer is in charge of devising and specifying $[S]$, a set of architectural and algorithmic solutions, from which worst-case strategies that can be deployed by "adversary" $< m.X >$ can be inferred and proved. Computer-based system S should always "win against" $< m.X >$. One of the greatest difficulties is to identify and prove these worst-case strategies.

A model Y dominates another model Z if the "adversary" reflected by model Y is less restricted than that reflected by model Z (or is equally restricted). Similarly, this definition of dominance or strength applies to architectural and algorithmic solutions, as well as to properties.

Examples of Model Classes

- Architectural models
- Definition of which are the modules to be architected.
- Centralized shared memory, distributed clients-servers interconnecting sensors and actuators, meshed topology of point-to-point links, multiaccess broadcast bus, group of replicated processors.

- External event types models
- Arrival model (for every type):
  periodic (*pr*), sporadic (*sr*), aperiodic (*apr*), arbitrary (*arr*); the latter is based on the concept of bounded arrival density, which is defined via a pair of variables, namely a sliding time window of some constant size (w) and a maximum number (a) of arrivals within any such window; unimodal arbitrary (*uarr*) is the model such that only one pair is defined (per event type); multimodal arbitrary (*marr*) is the general case (i.e., ordered multiple pairs for an event type).

$$marr \supseteq uarr \supseteq apr$$
$$marr \supseteq uarr \supseteq sr \supseteq pr$$

- Collective attributes:
  sets of event types that are mutually dependent (causally, chronologically), constraints on arrivals (e.g., time gaps) between different event types.

- System or computational models
  They model advance knowledge relative to upper and lower bounds on computational delays (e.g., time taken by a processor module to make a computational step) as well as on communication delays (e.g., time taken by a network module to transmit a message). More than 30 models known, organized in three sub-classes:
- synchronous (*sm*): bounds exist, and their values are known,
- partially synchronous (*psm*): bounds exist, but their values are unknown; or, postulated values hold only after some unknown time; etc.,
- asynchronous (*am*): bounds do not exist (even if bounds exist, they cannot be reached, even with infinite runs).

$$am \supseteq psm \supseteq sm$$

- Models of shared persistent variables
- Consistency sets, i.e. sets of variables whose values are bound to meet specific sets of invariants.

- External variables, which are shared between tasks and the environment, as well as variables that depend upon these external variables. Examples: variables shared between tasks and sensors, between tasks and actuators.
- Internal variables, which are shared among tasks only, and which do not depend on external variables.

• Task models
- Individual structure: *sequence*, *star*, *tree*, *finite graph*.

$$finite\ graph \supseteq tree \supseteq star \supseteq sequence$$

- Individual attributes (per task): external variables accessed, conditions for (re)activation, suspension, abort (e.g., arrivals of external or internal events), shared internal variables, roll-backs allowed or prohibited.
- Collective attributes: sets of causally dependent tasks, of chronologically dependent tasks.

• Failure models (of architectural modules)
- Value domain: incorrect values.
- Time domain: crash (crf) $\equiv$ permanent halt without warning, omission (omf) $\equiv$ correct steps are not taken from time to time, timing (tif) $\equiv$ steps are taken, but too early or too late.

$$tif \supseteq omf \supseteq crf$$

Many models can be built by combining both domains. More than 20 models known. Most restricted model is *stop* (either correct behavior or crash). Most unrestricted model is *byzantine* (any behavior in the value domain, in the time domain).

• Failure occurrence models (for every module)
See external event arrival models. Most often, the aperiodic or the arbitrary models are considered.

Examples of Property Classes

• Safety: every possible system run (a collective execution of tasks) satisfies some set of invariants. Examples are *mutual exclusion*, *serializability*, *data consistency*, *causal ordering*, *linearizability*.

$$consensus \supseteq causal\ broadcast \supseteq reliable\ broadcast$$

• Liveness: for every possible system run, tasks that are invoked are activated and make progress. Examples are deadlock freedom, collision detection-and-resolution in bounded time.

• Timeliness: tasks are assigned timeliness constraints. For every possible system run, every timeliness constraint is met.

A timeliness constraint belongs to a category and has a type.

- Examples of categories: latest termination deadline (ld), bounded jitter (bj), earliest start time (st).

$$st \wedge bj \supseteq st \wedge ld \supseteq bj \supseteq ld$$

- Examples of types: constant (c), linear function (lf), or non linear function (nlf) of system or environment parameters. An example of type lf would be $deadline = \alpha/temperature$. An example of type nlf would be $deadline = \beta.(altitude)^2$.

$$nlf \supseteq lf \supseteq c$$

- Dependability: in the presence of partial failures, every possible system run satisfies some set of invariants. Examples are:
- observability (*module*): states which type of failure behavior is to be exhibited by *module*.
- total order: histories of steps (e.g., reads and writes over replicated data) in a group of modules are uniquely ordered.
- availability (*service* [3]): states that *service* can be activated whenever requested.
- reliability (*service*): states that *service* must be correctly delivered.

Note that our definitions of availability, of reliability, are "free from probabilities". This is consistent with the views presented in section 2.3. Under given models - e.g., an upper bound f on the number of module failures that can accumulate without repair, a computational model - and for some given architectural and algorithmic solution, one can establish feasibility conditions such as, e.g., a lower bound on the number of modules needed to cope with up to f failures. Under such conditions, desired dependability properties hold for sure.

With traditional "stochastic" approaches, the definition of a dependability property involves a "measure" - e.g., a probability, a coverage factor - of the smallest acceptable likelihood that this property does hold. In fact, such "measures" come into play for the reason that some of the models considered may be violated - by the environment of S - at run time [4]. If we were to follow this line of reasoning, then safety, or liveness, or timeliness properties should also always involve some "probabilistic measure". As we know, this is not the case at all.

Again, we see no benefits deriving from having probabilities or coverage factors appearing in specifications of properties, on the ground that specified models/assumptions may be violated. Conversely, what does matter is knowing how to compute probabilities or coverage factors for a set of models (such as, e.g., the

[3] A fault-tolerant *service* is obtained via replicated tasks and data.

[4] Faulty implementations are another popular motivation. Besides the fact that this is tantamount to assume that there is fundamentally no hope of achieving correct implementations, one may wonder how such "measures" can be used a priori by, e.g., S/W engineers, as ladders of acceptable "faultiness".

conventional "up to f failures without repair" assumption - a failure occurrence model).

Even without embarking on probabilistic or statistical calculations, clients may want to specify how properties should be "degraded" if it happens that some models are violated at run time. This is often achieved by specifying - in addition to the property attributes presented above - individual task or service "value" functions and by requiring that, for every possible system run, accumulated "value" be maximized. This leads to optimization problems akin to, e.g., minimum regret.

Regarding timeliness, constant "values" have been considered [9]. Results of interest in the presence of general "value" functions can be imported from game theory or decision theory. Feasibility conditions may not be satisfied transiently, because of spurious violations of an event arrival model. Thanks to some appropriate algorithmic solution, tasks that do not meet their timeliness constraints precisely are those being the least valuable whenever such violations occur. Similar comments and solutions apply to dependability properties.

• Complexity of [$oracle.S$]

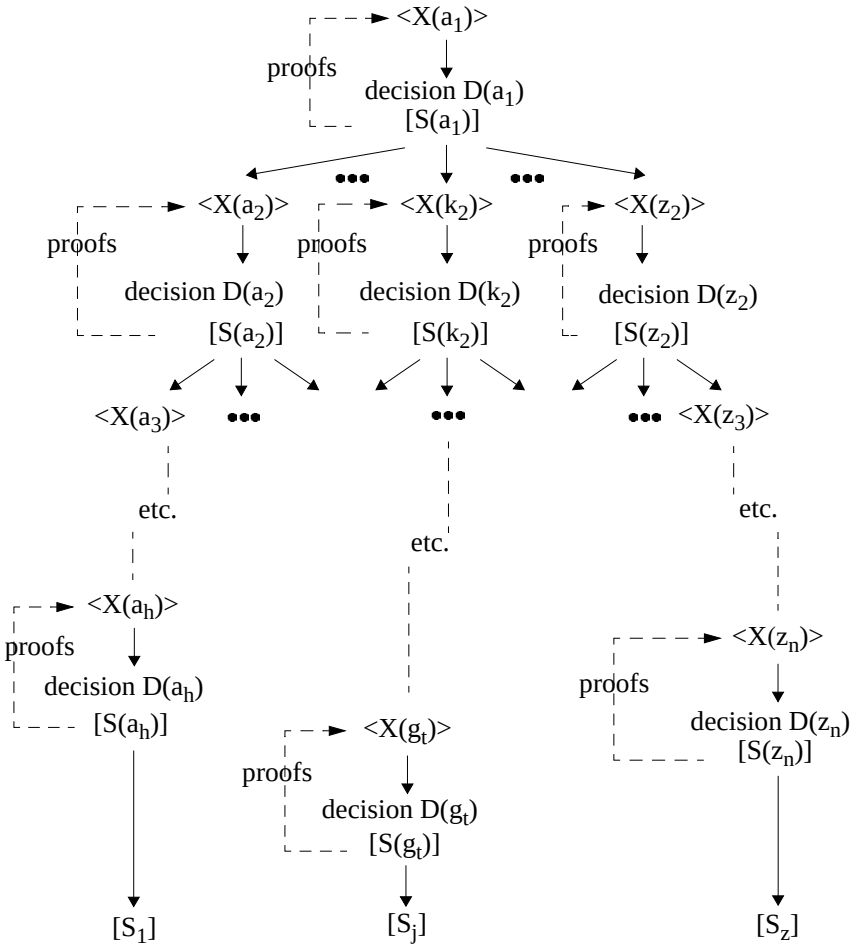
A client may want to bound *C(oracle)*, the complexity of a dimensioning oracle, which is determined by the complexity of feasibility conditions.

3.2 The System Design Phase

Overview

Most often, real problems $< X >$ are of exponential complexity. Consequently, from a practical standpoint, algorithmic solutions - meant to be used on-line by S - are sub-optimal in most realistic cases. Therefore, a prime contractor has many ways of constructing a design tree, for any given $< X >$. From a practical standpoint again, it may be required (by a client) - or it makes sense - to have some of the modules of S based on COTS products (e.g., processors, operating systems, middleware). Early adoption of COTS products is tantamount to "freeze" a priori some of the intermediate nodes or some of the leaves of a design tree. This reduces the space of potential solutions to be explored, which is a good idea at first sight. However, correctness proof obligations have to be met at any design stage, regardless of the fact that COTS products are or are not being considered at that stage. There are potential difficulties involved with COTS products (see further).

The tree structure of a design phase is shown figure 2, where $[S] = \cup_{j\in[1,z]}[S_j]$ is the (modular) specification of S that "aggregates" all design decisions. A prime contractor stops designing along a branch whenever the specification arrived at is known to be implementable (e.g., via some COTS product, or as a customized H/W and/or S/W module), or when the to-be-continued design work is assigned to a co/sub-contractor (who is bound to apply a proof-based SE method as well). Indeed, in practice, a prime contractor may decide to end a design tree branch "prematurely", i.e. when arrived at some non implementable specification module $[S_u]$. Design work - to be continued - is contractually handed over to some



$$\text{modular specification of } S = [S] = \bigcup_{j \in [1,z]} [S_j]$$

Fig. 2. The tree structure of a design phase

co/sub-contractor. As explained further, the specification of problem $< X' >$, which is the contract handed over by the prime contractor, is directly extracted from $[m.S_u]$. To a co/sub-contractor, $< X' >$ is a design tree root.

Design Stages

Each stage has a unique reference r_i, i being a tree level and r being one element of the set of names needed to enumerate all stages of the same level. Problem considered at stage r_i is denoted $< X(r_i) >$. Initial problem $< X >$ is rewritten $< X(a_1) >$. Design decision/solution or design stage r_i is denoted $D(r_i)$.

Generic solutions

Design stage $D(r_i)$ consists in dividing $< X(r_i) >$ into a number - denoted $d(r_i)$ - of modules or subproblems, by specifying a modular decomposition. For every such module, one specifies a set of models, denoted $[m_k(ri)]$, $k \in [1, d(r_i)]$. Given a modular decomposition, one specifies a global architecture - denoted $[arch(r_i)]$ - as well as a global algorithmic solution (a composite algorithm, in most cases) - denoted $[alg(r_i)]$. Instantiation of $[alg(r_i)]$ (resp. $[arch(r_i)]$) local to module k is denoted $[alg_k(r_i)]$ (resp. $[arch_k(r_i)]$). Choice of models $m_k(r_i)$ is constrained by the obligation of fulfilling a design correctness proof (see further). Furthermore, for some modules, it may be that some associated models depend on $alg(r_i)$. This is often the case with event arrivals models (events triggered among modules).

The output of design stage $D(r_i)$ is a triple $\{[S(r_i)], \mathcal{P}.S(r_i), [oracle.S(r_i)]\}$, where :

- $[S(r_i)]$ comprises the $d(r_i)$ sets $\{[m_k(r_i)], [alg_k(r_i)], [arch_k(r_i)]\}$,
- $\mathcal{P}.S(r_i)$ is the set that describes those global properties - along with their proofs - that hold with $[S(r_i)]$,
- $[oracle.S(r_i)]$ specifies a dimensioning oracle (see further) which is derived from set $\mathcal{P}.S(r_i)$.

- Specification $[S(r_i)]$

For each of the specification sets in $[S(r_i)]$, there is one out of two possible outcomes. Consider $[S_k(r_i)]$. If deemed implementable or implemented by some procurable product, that specification terminates a design tree branch. In other words, $[S_k(r_i)]$ is a (level i) design tree leaf, i.e. a specification of a real (physical) module of S. If deemed non implementable or not implemented by some procurable product, then $[S_k(r_i)]$ - which then specifies an abstract module of S - translates into a (level $i+1$) subproblem $\{< m.X(q_{i+1}) >, < p.X(q_{i+1} >\}$ trivially extracted from $[m_k(r_i)]$.

Let us illustrate the above with the following example drawn from our experience with project 1 (see section 4), where $< X(a_1) >$ is a distributed real-time fault-tolerant computer science problem and $[S(a_1)]$ is based - in particular - on a network module, node k of level 1 in the design tree. In $[m_k(a_1)]$, one finds the following :

- synchronous computational model : network delays range between min and max,
- failure model : bounded message omissions.

Furthermore, given the (external) event arrivals and the task models specified in $< m.X(a_1) >$ on the one hand, the algorithmic solution $alg(a_1)$ on the other hand, one derives time windows within which messages can be generated by tasks, i.e. the message arrivals model to be found in $[m_k(a_1)]$, e.g., the unimodal arbitrary model. Such derivations may raise non trivial issues, barely addressed so far in scientific publications.

Obviously, this postulated network module is not directly implementable with existing COTS products, or with other documented products. In fact, $[m_k(a_1)]$ raises the following level 2 subproblem (say, tree node c_2) :

- $<$ $m.X(c_2)$ $>$ $=$ $<$ unimodal arbitrary message arrivals model, message omission failure model $>$,
- $< p.X(c_2) > = <$ network delays $\in$ [min, max] $>$.

Solving this subproblem entails - in particular - designing a real-time communication protocol (an algorithmic solution) and establishing timeliness proofs, such as done in [4].

• Set $\mathcal{P}.S(r_i)$

This is the set describing properties and proofs that properties "as strong as" those specified in $< p.X(r_i) >$ do hold with $[S(r_i)]$. Establishing the existence of such properties is needed so as to meet a design correctness proof obligation, i.e. to prove that $[S(r_i)]$ is a correct solution. This is also essential for establishing $[f\_oracle.S]$ (see further). Proofs in $\mathcal{P}.S(r_i)$ may rest on establishing computable behavioral functions such as, e.g., upper (resp. lower) bounds on task response times (denoted B (resp. b)), lower bounds on redundancy degrees (denoted R).

• Specification $[oracle.S(r_i)]$

Most often, $[S(r_i)]$ contains unvalued variables, specified as per $[s(r_i)]$, such as, e.g., a boolean matrix reflecting how level i S/W modules are mapped onto the modules considered with $[S(r_i)]$, periods of invocation for task schedulers over every such module. $[s(r_i)]$ is decomposed as follows :

- $[!s(r_i)]$, which contains variables that may be valued (by a client or a prime contractor) prior to conducting a dimensioning phase; such valuations are performed via a tool component denoted $configurator.S(r_i)$; for example, some mappings being geographically constrained (for some particular release of S), corresponding variables have pre-determined values; some of the variables in $[!s(r_i)]$ may be left unvalued,
- $[?s(r_i)] = [s(r_i)]$ - $[!s(r_i)]$, which contains variables to be valued by a tool component denoted $f\_oracle.S(r_i)$, i.e. by running (dimensioning) $oracle.S$ (see section 3.3).

In $[f\_oracle.S(r_i)]$, one finds generic feasibility conditions; such conditions are a set of constraints that, for pair $\{< X(r_i) >, [S(r_i)]\}$, bind together variables that appear in $< m.X(r_i) >, < p.X(r_i) >$, set $\mathcal{P}.S(r_i)$ and $[m.S(r_i)] = \bigcup_{k\in[1,d(r_i)]}[m_k(r_i)]$. Let $[oracle.S(r_i)]$ be the union of $[configurator.S(r_i)]$ and $[f\_oracle.S(r_i)]$.

Let us now take a broader view, and consider a complete design phase. As shown fig. 2, design tree leaves are relabelled consecutively. As for $[S]$, we have $[oracle.S] = \bigcup_{j\in[1,z]}[oracle.S_j]$, which is to be implemented as a tool component. Note that, rigorously speaking, $[S]$ is only partially implementable. For example, algorithms and protocols can be implemented in some language (or reused, if available), without having to know the outcome of a dimensioning phase. However, full implementation of S is conditioned on knowing $V([s])$.

Design correctness proof obligation: design_cpo($D(r_i)$)

A generic solution $D(r_i)$ provably solves problem $< X(r_i) >$ if the following two conditions are satisfied:

$$p.S(r_i) \supseteq p.X(r_i) \quad \text{and} \quad m.S(r_i) \supseteq m.X(r_i),$$

where $p.S(r_i)$ are properties enforced by $[S(r_i)]$, as given in $\mathcal{P}.S(r_i)$.

Obligation design_cpo($D(r_i)$) expresses the constraint that properties ensured with some $[S(r_i)]$ must dominate those stated in $< p.X(r_i) >$, under models that must dominate those stated in $< m.X(r_i) >$.

For example, with the real-time problem sketched out in section 2.2 (system design phase), picking up the periodic arrival model, which is central to the rate/deadline monotonic approach, would be invalid in our case. Indeed, proofs that the first condition of design_cpo(.) is met (no deadline is missed) would be irrelevant, as being non applicable to the problem under consideration (revealed by a violation of the second condition).

In fact, the periodic arrival model is quite inappropriate for modeling event arrival laws within distributed systems (i.e. its coverage factor is close to 0), even if one would assume periodic models for external event arrivals. This is essentially due to the fact that task durations, resource contention delays, sojourn times in waiting queues, are inevitably variable, even more so when failures are considered. Consequently - and contrary to widespread belief - results on periodic task scheduling have very limited usefulness in the case of distributed computer-based systems, even more so when established for sequential task models.

COTS Products and Current SE Practice

As stated earlier, one may have to conduct a design phase under the constraint that COTS products have been selected a priori. A COTS product has an architecture and contains inner mechanisms (algorithms) that ensure specific properties, for some environments. A vendor may not want to reveal which algorithms are used in a product that is considered for implementing some design tree leaf (say q), nor the proofs of properties. Keeping $\{[S_q], [s_q]\}$ undisclosed is acceptable provided that there is a commitment on a pair $\{< X >, < x >\}$ that specifies which is the problem solved by that product, and that there exists a

matching dimensioning oracle. Under current industry/business practice, COTS products are not accompanied with (provably correct) specifications, neither with oracle-like tools. Consequently, there is no way of checking whether early adoption of a COTS product is or is not a correct design decision, or whether - if correct - that design decision is arbitrarily non optimal (hence, costly). Which explains many of the problems that bring clients and prime contractors, or prime contractors and co/sub-contractors, into conflict.

Yet, it is easy to eliminate such problems simply by requiring that design correctness proof obligations be fulfilled at every design stage, including those based on COTS products. Of course, it will take some time before we see S/W, computing or networking companies ship COTS products along with such specification sets as $\{[S_q], [s_q]\}$ or $\{< X >, < x >\}$. This simply mirrors the current immaturity of the (relatively young) computing and information industry. Is it not surprising that vendors of refrigerators make contractual commitments on specifications akin to $< A >$ or $< X >$, whereas vendors of such COTS products as "real-time" operating systems or middleware do not publish or do not make contractual commitments on similar specifications?

As explained earlier, fulfilling design correctness proof obligations yields feasibility conditions (FCs). One may establish FCs without expressing computable behavioral functions such as b, B or R. For example, for some {uniprocessor, periodic arrival model, timeliness} problems, it has been proved that Earliest-Deadline-First is an optimal algorithm, FCs being stated as $U \leq 1$, where U is a processor utilization ratio. Latencies (i.e., slack times before deadlines) of periodic tasks are unknown. One may also establish FCs by first expressing B (in $\mathcal{P}.S$), and then constrain it to match the timeliness property specified in $< p.X >$ by fulfilling a design_cpo. One would then know task latencies.

With realistic problems, the number of scenari that can be deployed by any given "adversary" $< m.X >$ is "high". "High" also is the number of states a computer-based system might enter whenever any of these scenari is instantiated. Consequently, set θ of pairs {adversary scenario, system state} may not even be enumerable (think of partial failures). FCs are generic computable constraints. They demarcate the boundary of a set of scenari that includes all worst-case scenari that can be instantiated by "adversary" $< m.X >$. This boundary determines a subset θ_f of set θ.

Under current SE practice, identification of set θ_f is ignored. A computer-based system S is accepted by a client provided that S has successfully gone through a finite number of tests that are supposed to represent real future operational conditions. Definition of these tests usually is based on past experience and on the identification of specific pairs {adversary scenario, system state} that are considered to be "of particular concern". This is tantamount to invite a client - who should not be involved at all in such decisions - to make early commitments on a limited number of elements in set θ, without knowing whether one is testing for worst-case scenari nor how many of worst-case scenari are left untested.

This explains why simulation and integration testing, as performed under current SE practice, do not allow for verifying whether or not a specification

$\{[S], V([s])\}$ is satisfied, even less a specification $\{< X >, V(< x >)\}$. Which explains many operational failures and project cancellations.

Conversely, it is reasonably obvious that FCs bring along the following benefits: (i) FCs yield a dimensioning of S such that problem-centric properties quantified as per $V(< p.x >)$ certainly hold for all possible worst-case quantified scenari "hidden" within $V(< m.x >)$, (ii) time needed to compute a set of constraints along the boundary of θ_f is vastly smaller than checking whether some invariants are satisfied for every element in θ_f, which is what must be done if one would resort to event-driven simulation or model checking (assuming furthermore that θ_f has been identified under these approaches). See sections 3.4 and 3.5 for more on this issue.

Final Comments

Experience shows that early design stages - design stage 1 in particular - are conducted more or less empirically under current SE practice. This explains many of the setbacks and operational failures experienced over the last 10 years with projects involving complex and/or critical embedded systems.

An appropriate illustration of this reality is the US FAA AAS project (air traffic control). The Center for Naval Analyses, which was asked to audit the project, diagnosed, in particular, the following problems and oversights:

- "There is no evidence that any in-depth analysis of the design architecture or the contractor methodology was conducted."
- "Major system and architectural engineering, so desperately needed early in the program, was not evident."
- "In addition to the S/W problems, the system lacked overall system engineering. The S/W and the H/W were broken into pieces without looking at the overall system requirements."

Another appropriate illustration of this reality is the failed maiden flight of the Ariane 5 European satellite launcher (see section 5).

See section 4 for an example of how to conduct design stage 1 in the case of a (complex and critical) modular avionics problem.

3.3 The System Dimensioning Phase

This phase can be conducted by a client, a prime contractor, or both. Before an implementation or some specific release of S can be undertaken, the physical dimensioning of every module of S need be specified. Such a dimensioning $V([s])$ depends upon a quantification specification $V(< x >)$. Recall that one cannot proceed with fully implementing or releasing S without knowing $\{[S], V([s])\}$.

Specification $[s]$ is the union of those subsets that contain unvalued system variables accumulated throughout design stages, i.e. from the root of a design tree. Hence, as for $[S]$, $[s] = \cup_{j \in [1,z]}[s_j]$ is a modular specification. As seen before, $[s]$ is decomposed as follows:

- $[!s_j]$, which contains variables that may be valued prior to conducting a dimensioning phase,

- $[?s_j] = [s_j]$ - $[!s_j]$; values to be assigned to variables in $[?s_j]$ are obtained as a by-product of verifying that specified feasibility conditions are satisfied.

Consequently, dimensioning oracle *oracle.S* for pair $\{< X >, [S]\}$ includes the following:

- a configuration component, denoted *configurator.S*; its input is some valuation of some of the variables in $[!s]$; its output is $V([!s])$,
- a feasibility oracle, denoted *f_oracle.S*, a component which implements feasibility conditions established for pair $\{< X >, [S]\}$; its inputs are $V([!s])$ and $V(< x >)$; its output is $V([?s])$.

For example, some valuation is assigned to a boolean matrix in $[!s]$ that reflects how application-level (i.e. level 1) S/W modules are to be mapped onto (real) modules of S. Values of other variables in $[!s]$, e.g. matrices that give mappings of low-level S/W modules, are computed by $configurator.S$.

Examples of variables that are valued by *f_oracle.S* are as follows in the case of a processor module: exact invocation period of a task scheduler, maximum execution time of this scheduler, exact or sufficient memory capacity needed to store the data structures used by application S/W modules mapped onto that module, exact or sufficient size of every buffer needed to store shared external variables and related variables, exact or sufficient size of every waiting queue, exact values of timers used to detect failures, lower bound of the group size that module belongs to. For a communication bus module, one would have: number of access ports, number of independent physical channels, exact or sufficient sizes of waiting outqueues and inqueues (one of each kind per access port), and so on.

Component *f_oracle.S* works as follows. For every variable in $[?s]$, it selects a potential value within a given range (ranges are specified in the feasibility conditions) and iterate in this range until, either quantification $V(< x >)$ of problem $< X >$ is declared non feasible, or a value that meets the dimensioning_cpo is found.

Let $V([s])$ be some proposed dimensioning of $[S]$. How do we know whether $V([s])$ is correct w.r.t. problem quantification $V(< x >)$?

Dimensioning correctness proof obligation: dimensioning_cpo($V([s])$)
A system dimensioning $V([s])$ provably satisfies a quantification $V(< x >)$ of problem $< X >$ if the following two conditions are satisfied:

$$V([p.s]) \supseteq V(< p.x >) \text{ and } V([m.s]) \supseteq V(< m.x >).$$

An illustration of the first condition for a real-time problem stating *bounded jitters* as a timeliness property would be that values of behavioral functions b and B, computed considering some processors speeds and processors storage capacity, match the values of bounded jitters set via $V(< p.x >)$. An illustration of the second condition would be: these values have been computed for values of upper bounds on arrival densities at least equal to those stated via $V(< m.x >)$.

It is reasonably obvious that whenever a proof-based SE method is resorted to, meeting a dimensioning_cpo with some $V([s])$ is automatically enforced by a

dimensioning oracle, given that values of variables in $[?s]$ are computed by the oracle. Nevertheless, it may be that meeting such a cpo is not possible, in which case the oracle returns a negative output. This may happen essentially for two reasons. One is that $V(< x >)$ is too "ambitious" (e.g., very small deadlines under very high arrival densities and for a widely dispersed architecture). Another (practical) reason is that acceptable $V([s])$'s are constrained, either technology-wise (there is no processor running at the speed required to meet $V(< s >)$) or budget-wise (their is a limit on what a client is willing to spend on S).

Another motivation behind the concept of a dimensioning_cpo is linked with the fact that proof-based SE is not very much practiced yet. Hence, especially when COTS products are considered, system engineers should be aware of what is involved with correctly dimensioning a computer-based system. Whenever COTS or proprietary products are targeted for the implementation of S, values of some variables in $[!s]$ and $[?s]$ are "frozen" a priori, i.e. prior to entering a design phase, which increases the likelihood of delivering a system S that will fail when being used. This can be avoided by showing system engineers how to check whether a computer-based system is correctly dimensioned. Quite often, approximate over-dimensioning - which is commonly practiced, to clients' expenses - may be shown to violate dimensioning_cpo's.

Correct $V([s])$ is optimal for the solution specified in $[S]$ if, by choosing a dimensioning "weaker" than $V([s])$, at least one of the conditions of the dimensioning_cpo is not satisfied. Physical dimensionings translate into costs. In other words, with optimal $V([s])$, one knows what is the cheapest correct S that satisfies $\{< X >, V(< x >)\}$, with the solution considered.

Many different problem quantifications $V(< x >)$ may be contemplated by a client or a prime contractor, before a decision is made, e.g. before the fielding of a particular instantiation of S, or each time a new mission is set up. The latter would apply, for example, with embedded systems used for defense applications. Depending on the application considered, a client may or may not be willing to restrict complexity $C(oracle)$ - see section 3.1. Most real problems being of exponential complexity, an optimal oracle is hardly usable in practice. Small (resp. big) values of $C(oracle)$ translate into fast (resp. slow) oracles, but also into pessimistic (resp. more accurate) oracles. A pessimistic oracle may return a negative output despite the fact that problem $\{< X >, V(< x >)\}$ is feasible.

Note that the dimensioning of every variable that is shared by S and its environment (i.e., every external variable) is provably correctly specified, rather than being left undefined or to be decided upon by implementors of system engineering decisions (e.g., S/W engineers). Finding a correct dimensioning of a variable internal to some application S/W module is under the sole responsibility of the S/W engineer(s) in charge of that module. Conversely, finding a correct dimensioning of an external variable that is used by some application S/W module is not a S/W engineering issue.

Failure to understand or to acknowledge this distinction usually leads to disasters, as illustrated by the unsuccessful maiden flight of the European Ariane 5 launcher (see section 5).

Final Comments

As discussed in section 2.3, with deterministic proof-based SE, one may have to estimate coverage factors for some of the models specified in $< m.x >$, as well as for some of the quantifications specified via $V(< m.x >)$ and $V(< p.x >)$. By the virtue of the design_cpo's and dimensioning_cpo's, coverage factors related to $\{[S], V([s])\}$ can only be greater than those estimated for problem-centric set $\{< m.X >, V(< x >)\}$. Consequently, the capacity for a provably correct S to satisfactorily cope with violations of $\{< m.X >, V(< x >)\}$ is higher than mirrored by the problem-centric coverage factors, this being obviously true whenever feasibility conditions are sufficient rather than necessary and sufficient.

3.4 Benefits achieved with Proof-Based System Engineering

In addition to avoiding system engineering faults, a number of advantages derive from proof-based SE. Some of them are briefly reviewed below.

Technical Benefits

- Design reusability. Various generic application problems $\{< A >, < a >\}$ may translate into the same set $\{< X >, < x >\}$. Whenever a generic problem $< X >$ has been provably solved in the past, matching generic solution $[S]$ comes for free for any of these application problems. (Potentially huge savings in projects durations and costs).
- High and cheap system configurability. The fact that system design and system dimensioning are distinct phases makes it possible for a client to select at will any combination of application S/W modules, some quantification of the application problem of interest, and to be delivered a correctly dimensioned system S, without having to incur any of those delays and costs that are induced by the need to re-design S.
- Elimination of artificial dependencies between the capture, the design and the dimensioning phases has the effect of suppressing time and budget consuming inefficient work, such as looping back and forth between different design stages or between different phases of a project lifecycle.
- Rigorous assessment of COTS technology, via design and dimensioning correctness proof obligations.
- Simplification of final integration testing. Verification that a set of concatenated modules "behaves correctly" - the "system integration phase" - runs into combinatorial problems. Under current practice, such a verification is done via testing, within imposed bounded time and budget, which means that testing is necessarily incomplete, even with computer-based systems of modest complexity. Let us consider conservative figures. Imagine that 10 modules, each having 100 visible states (reduction to such a small number being achieved via unitary testing), go through integration testing. Complete testing involves checking 10^{20} global states, which is beyond feasibility. (Even if a global state could be checked in 1 millisecond, complete testing would take in the order of 3.10^9 years). Under a proof-based SE approach, it is proved beforehand that the union of modules is a global solution. If it would

be the case that every specification pair $\{[S_j], V([s_j])\}$ is correctly implemented, there would be no need for system S integration testing (theoretical viewpoint). Unitary tests, i.e. on a per module basis, being incomplete in general, system S integration testing still is necessary (practical viewpoint). However, the beneficial effect of proof-based SE is to bring the complexity of system integration testing down to acceptable figures, such as, e.g., pseudo-linear complexity in the number of modules' visible states.

Legal Benefits

Would a problem arise during the implementation, construction, fielding or utilization of a system, it is easy to find out which of the actors involved is/are to be held responsible. Contrary to usual practice - actors involved share the penalties - it is possible to discriminate without any ambiguity between client's fault (faulty specification set $\{< A >, < a >\}$), prime contractor's fault (faulty specification sets $\{< X >, < x >\}$, $\{[S], [s]\}$, or $[oracle.S]$), co/sub-contractors' fault (faulty implementation of set $\{[S], [s]\}$), prime contractor's or some co/sub-contractor's or some tool vendor's fault (faulty implementation of $[oracle.S]$).

Strategic Benefits

It is common practice to assign different pieces of a project to different contractors. It is possible for a client or a prime contractor (say M) to take full responsibility w.r.t. a complex application problem, to "break" it into simpler subproblems, either in one design stage or until some appropriate level of problem granularity has been reached. Subproblems can then be contracted, each having a precise specification $\{< X(.) >, < x(.) >\}$. M may ask contractors to show proofs of design and dimensioning correctness. Furthermore, note that M only has complete knowledge of the overall technical decisions. Therefore, M may subcontract work to its competitors, without giving away its technical know-how.

Of course, there are also many new (strategic) opportunities for tool vendors as well. Knowledge-based tools needed to support the various phases of proof-based SE are yet to be developed.

3.5 Proof-Based System Engineering and the Software Crisis

Our main thesis is that one of the major causes of the so-called S/W problems or "S/W crisis" is lack of proper identification of the real nature of those problems that the S/W industry has been faced with for years, without succeeding in solving them. Complexity is an archetypal example. In many instances, these problems are system engineering issues. It is not surprising at all that SE issues do not fade away when addressed as S/W engineering issues. Trying to resolve them by improving the quality of S/W implemented modules is like trying to successfully build a dam by improving the quality of blocks of reinforced concrete, without realizing that the problems are due to incorrect dam blueprints.

Whenever an operational computer-based system fails, S/W is being run at time of failure. It is then all the more tempting to believe that S/W is to be held

responsible for a failure. Unfortunately, just because a problem is observed in S/W behavior does not mean that it is a S/W problem per se. Many such problems, especially the more difficult and subtle ones, are rooted into SE decisions (problems originate in SE faults), and propagate to the S/W levels through deficient specifications. Audits of a number of failed projects have indeed revealed that causes of failures were not due to latent faults in application S/W, as was believed by clients and/or contractors.

Planned or delivered systems did not operate properly simply because they did not include appropriate "system-level" (e.g., scheduling, synchronization) algorithms. As a result, application S/W modules (programs) could be interrupted at random and/or could run interleaved in arbitrary fashion. Even with perfectly correct application S/W modules, behavior of such computer-based systems could only be incorrect.

One pre-requisite for solving the "S/W crisis" is to move away from monolithic application S/W. Modularization and S/W modules reuse are sound principles. However, in order to reap the benefits of such principles, one must show that quasi or truly parallel asynchronous executions of S/W modules cannot jeopardize the integrity of any of these modules (a property commonly found in $< A >$). Which translates into showing that a set of invariants (I) is always satisfied, i.e. into specifying a safety property in $< X >$, such as, e.g., linearizability or serializability [1], which raises problems that have numerous and well documented solutions.

One approach - which is at the core of some formal methods popular in the S/W engineering field - consists in exploring every global state that can be entered by a given set of application S/W modules, and to verify for each global state that (I) is not violated. Let us make some rough calculations. Consider a set comprising 100 S/W modules (a conservative figure) and assume that, in average, every single module can enter 10 different intermediate states that are visible from other modules (again, a conservative figure).

This entails searching 10^{100} global states, which is beyond tractability, even if one would reduce this complexity by a few orders of magnitude via, e.g., binary decision diagrams. Verification methods directed at parallel synchronous or asynchronous computations in the presence of failures - the correct paradigms with embedded systems - and which are based on exhaustive searches of global state spaces, are doomed to fail even with modestly complex systems. Furthermore, their usage is hardly justifiable, given that there exists an impressive gamut of fault-tolerant and/or concurrency control algorithms. Companion proofs establish that a system equipped with any of these algorithms guarantees that (I) cannot be violated, whatever the set of application S/W modules considered. Proofs do away with the need for verification. Algorithms (SE originated solutions) serve the purpose of "breaking complexity", in a very cost effective manner.

The above is a simple illustration of the fact that oversight of SE issues inevitably complicates S/W design and S/W development. Proof-based SE promotes S/W reuse, S/W evolution and facilitates S/W management.

4 Lessons Learned with Proof-Based System Engineering

In this section, we give examples of projects where the TRDF method has been applied. As indicated in section 2.4, the TRDF method is aimed at application problems that raise any combination of real-time, distributed, fault-tolerant computing issues. It can be demonstrated that whenever either one of these issues is raised in $< A >$, this translates into a problem $< X >$ where at least one of the other two issues is raised as well.

In projects 1 and 2, INRIA's REFLECS team took on the role of a prime contractor. For the sake of conciseness, presentations of projects will mainly consist in summarizing lessons learned, to the exception of two projects that will be more detailed, namely project 1 (see below) and project 4 (see section 5). Work conducted during these projects has not been published, to the exception of project 4. Reports, in French, are available upon request.

4.1 Project 1

Clients: French DARPA (DGA/DSP) and Dassault Aviation. Modular avionics was the application problem considered. Objectives were to check the feasibility of decoupling fully the capture, design and dimensioning phases, as well as to deliver specifications $[S]$ and $[oracle.S]$, that were subsequently implemented by Dassault Aviation. Practical constraints were that COTS products had to be taken into account (e.g., specific processors, "real-time" monitors, object-based middleware), and that one design stage only was to be undertaken.

Sketch of application invariant $< m.A >$

- Application S/W is modular; dates of creation of application S/W modules (set M of modules) span over many years. Modules can be suppressed or created at will. That is, set M is unbounded.
- Application S/W modules should not depend on any existing or future COTS products.
- External events are pilot commands, sensors data, etc. They trigger application S/W modules which, when executed, produce outputs, such as responses displayed to pilot, commands applied to actuators.
- Numbers of sensors, actuators, are finite but unknown.
- Persistent variables that represent current global plane, environment, and system states, are read and written when application S/W modules are run.
- Any such variable may be shared among some unknown number of S/W application modules, as well as among modules and the environment.
- There should be no restrictions on the programming models used to develop application S/W modules; no restrictions either on which variables can be accessed by a S/W module.
- Distribution of application S/W modules and shared persistent variables (over future embedded system S) should be unrestricted.
- Some external event types are periodic sensor readings, arrival laws for other event types (a majority) are unknown.

- Failures internal to (future) S should be anticipated (environment is "aggressive").
- Failures occur at unpredictable times.

Sketch of application invariant $< p.A >$

- Many different releases of M, involving any combination of application S/W modules, can be fielded; this should not entail re-designing or re-proving S.
- At all times, any application S/W module should run as expected (no side-effect).
- At all times, values taken by shared persistent variables should consistently mirror the current plane and plane's environment states.
- Outputs generated by application S/W modules are non recoverable.
- Application S/W modules must meet "hard" real-time constraints.
- Some of the persistent variables are critical. In case such a variable is lost, pilot should be informed "rapidly".
- One should be able to generate any particular release of S (proved correct) "rapidly".

Sketch of problem invariant $< m.X >$

- The executable counterpart of an application S/W module is a task.
- Any task t, if run alone, has an upperly bounded execution time.
- Any set T of tasks t can be considered; size of T is unknown.
- Architectural model: a distributed system of modules, no shared memory; number of modules is unknown.
- External event type models: Set EV of event types (ev). Mapping of EV onto T to be specified via a boolean matrix. Subset EV1: periodic arrival model (values of periods: unknown). Subset EV2: unimodal arbitrary arrival model (values of densities: unknown).
- Computational model: synchronous.
- Models of shared persistent variables:
  - Consistency sets in set DATA, i.e. subsets of variables whose values are bound to satisfy client defined invariants (I). Any task, if run alone, satisfies (I). There are no restrictions on which invariants (I) can be considered.
  - Set c.DATA = subset of DATA; defines the set of "critical" variables.
  - Type "internal" or "external" assigned to every shared persistent variable.
- Mapping of DATA onto T to be specified via a boolean matrix.
- Task models: finite graphs.
- Failure Models:
  - Processor modules: stop.
  - Network module: omission.
- Failure occurrence model: aperiodic and unimodal arbitrary arrival model.

Sketch of problem invariant $< p.X >$

- Safety:
  - Exactly-once semantics for tasks. No task roll-backs.

- Invariants (I)
- Serializability: every possible run for any pair {T, DATA} satisfies (I).

• Timeliness:
Task timeliness constraints: latest termination deadlines. Values of individual deadlines are unknown.

• Dependability:
- Very high availability and reliability for network services.
- Finite failure detection latency for computer modules that host c.DATA.

• Complexity $C(oracle)$: pseudo-linear in the number of tasks in T.

It turns out that $< X >$ specifies a generic computer science problem that is at the core of such drastically diverse application problems as stock markets, currency trading, defense (C^3I, space), air traffic control, nuclear power plants.

Problem $< X >$ is NP-hard. The algorithmic solution alg, specified in delivered $[S]$, includes a hybrid algorithm that is a combination of periodic distributed agreement, idling and non-idling, preemptive and non-preemptive First-In-First-Out, Earliest-Deadline-First, and template-based schedulers.

Sketch of $oracle.S$: precomputed schedule templates (not precomputed schedules) and particular constructs on graphs helped reduce $C(oracle)$ to what was specified. Note that the combination of arbitrary arrival models with finite graph task models yields a reasonably unrestricted "adversary". It turns out that, under such models, those proof techniques traditionally used to establish timeliness properties for weaker models (e.g, sequential tasks and periodic/sporadic arrival models) simply do not apply. Examples of techniques and results that were used are: optimal scheduling algorithms, analytical calculus, time based agreement algorithms, adversary arguments, matrix calculus in (max,+) algebra [2].

Lessons learned

The genericity of the TRDF method has been validated. It has been verified that it is possible to decouple fully the capture, design, and dimensioning phases. It was confirmed that the capture phase is essential and that having a contract such as $\{< A >, < X >\}$ is instrumental in clarifying responsibilities. At one point, Dassault Aviation felt that solution alg "had a problem". It did not take us long to agree that alg was correct vis-à-vis the contrat specified. What this client had in mind was in fact a variation of $< A >$. It turned out that the variation could be accomodated by alg, by restricting potential choices w.r.t. the schedule templates. However, the feasibility oracle (in $[oracle.S]$) had to be partially revisited.

What has also been clearly verified is how fast a dimensioning oracle can be compared to an event-driven simulator. For the task sets considered, run times never exceeded one hour with our dimensioning oracle, whereas run times could be as long as one day for the event-driven simulator developed by Dassault Aviation.

4.2 Project 2

Client: Institut de Protection et de Sûreté Nucléaire, French Atomic Energy Authority. The problem we had to examine was whether some COTS command-and-control system could be considered for automated safety related control operations in nuclear power plants.

We applied the TRDF method twice, in parallel. One thread of work was concerned with the capture phase only. Upon completion, the client was delivered specifications $< A >$ and $< X >$.

The other thread of work consisted in doing what can be called reverse proof-based system engineering. The COTS system considered had no such specification as $\{[m.S], [p.S]\}$ or $[alg]$. Consequently, we had to inspect the technical documentation and get information from engineers who were familiar with this system. The TRDF method was applied bottom-up. By analyzing the modules of the COTS system, we could reconstruct $[m.S]$ and algorithmic solutions of interest alg, which led us to specification $[p.S]$.
We could then establish that none of the conditions of a design-cpo was fulfilled (i.e., $[S] \supseteq < X >$ did not hold true). Our recommendation has been to disregard the command-and-control system under study.

Lessons learned

Here too, it was verified that, when conducted rigorously, a capture phase is time consuming. One difficulty we ran into during the first iterations was to have the client's engineers "forget" about the technical documentation of the COTS system they were familiar with, and to concentrate on their real application problem. The other lesson is that it is indeed possible to rigorously assess COTS products.

4.3 Project 3

Clients: French Ministry of Research and Dassault Aviation. We had to identify what were the dependability properties actually enforced by a simple embedded system considered for commanding and controlling essential actuators in future planes. Different teams of S/W engineers and scientists from various French research laboratories had applied formal (proof-based) methods to specify, develop and engineer S/W needed to command and control the actuators. When our work was started, proofs of S/W correctness had been established. Our work in this project was a bit similar to that of project 2. The TRDF method allowed to show that, despite redundant S/W and a redundant architecture, the embedded system could fail under certain circumstances. Essentially, that was due to the fact that physical time is heavily relied upon in order to achieve necessary synchronizations, and that physical time may not be properly maintained in the presence of partial failures.

Lessons learned

What was revealed by applying the TRDF method was the exact "gap" existing between those models (e.g., system model, failure models, event arrivals

models) assumed in order to establish S/W correctness proofs and those models that correctly reflect the real embedded system and its environment. S/W correctness proofs were developed assuming models much weaker (in the $\supseteq$ sense) than the correct ones.

This is an illustration of the fact that it is not a good idea to apply formal S/W engineering methods without applying proof-based system engineering methods as well. This is due to the fact that those models that can be accomodated with existing formal S/W engineering methods are ideal representations of physical reality and of real computer-based systems. Hence the following choices: (i) either ignore such limitations, which inevitably leads to operational failures, (ii) or acknowledge these limitations and conclude - erroneously - that S/W correctness proofs are useless, (iii) or acknowledge the need for proof-based SE, whose role is to "bridge the gap" between both worlds. In other words, one of the benefits of proof-based SE is to emulate over real systems those "simple" models that can be currently accommodated with formal S/W engineering methods.

Project 4 - described in section 5 - is concerned with how the TRDF method was applied to diagnose the causes of the failure of Ariane 5 Flight 501.

5 A Case Study in Proof-Based System Engineering: The Ariane 5 Flight 501 Failure

5.1 Introduction

On 4 June 1996, the maiden flight of the European Ariane 5 satellite launcher ended in a failure, entailing a total loss (satellites included) in the order of 0.5 Billion US dollars and a delay for the Ariane 5 program, which was initially estimated to be in the order of 1 year. An Inquiry Board which was formed by the European Space Agency (ESA) - the client - and the French Space Agency (CNES) - the prime contractor - was asked to identify the causes of the failure. Conclusions of the Inquiry Board were issued on 19 July 1996, as a public report [6]. The failure analysis presented in the sequel is based upon the Inquiry Board findings.

Our conclusions deviate significantly from the Inquiry Board findings. Basically, the Inquiry Board concludes that poor S/W engineering practice is the culprit, whereas we argue that the 501 failure results from poor system engineering decisions. Most of what is labelled as S/W errors are in fact manifestations of system engineering faults.

Had a proof-based SE method been applied by the prime contractor and by the industrial architect (Aérospatiale), faults that led to the 501 failure would have been avoided. This analysis is meant to - hopefully - help those partners in charge of and involved in projects similar to the Ariane 5 program. Neither the prime contractor's system engineers nor the industrial architect's system engineers should be "blamed" for not having followed a proof-based SE approach, given that proof-based SE methods did not exist at the time the Ariane 5 onboard embedded system was ordered and designed.

Conversely, it would be hardly understandable to keep "taking chances" with such grandiose projects now that the basics of proof-based SE have emerged. No changes in system engineering methods or exclusive focus on S/W engineering issues is what we mean by "taking chances". Hopefully, this analysis may also serve to demonstrate that it is inappropriate to "blame" the S/W engineers involved, i.e. engineers with the industrial architect and with sub-contractors Matra Marconi Space and Sextant Avionique.

An initial version of this analysis appeared in [13].

5.2 The Failure Scenario

The Ariane 5 flight control system (FCS) has a redundant architecture. Two identical computers (SRI1 and SRI2), running identical S/W, are responsible for extracting inertial reference data (launcher attitude) and for periodically sending two other identical flight contol computers (OBC1 and OBC2) messages that contain flight data. A redundant MIL-1553-B bus interconnects the SRIs and the OBCs. The OBC computers, which run identical S/W, are in charge of maintaining the launcher on its nominal trajectory, by commanding the nozzle deflections of the solid boosters and the main engine. The FCS redundancy management is of type passive: in the SRI logical module, in the OBC logical module, only one physical module is active at any time, the other one - if not failed - being the backup.

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 boosters and the main engine. These deflections were commanded by the S/W of the OBC active at that time, whose input was data transmitted by the SRI active at that time, that is SRI2.

Part of these data did not contain proper flight data, but showed a diagnostic bit pattern of SRI2, which was interpreted as regular flight data by the OBC. Diagnostic was issued by SRI2 as a result of a S/W exception. The OBC could not switch to the backup SRI1 because that computer had ceased functioning 72 ms earlier, for the same reason as SRI2.

The SRI S/W exception was raised during a conversion from a 64-bit floating point number N to a 16-bit signed integer number. N 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 computers).

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 the alignment task. This task serves a particular purpose after lift-off with Ariane 4 but is useless after lift-off in the case of Ariane 5.

5.3 Conclusions and Recommendations from the Inquiry Board

According to the Inquiry Board, causes of the 501 failure are S/W specification and S/W design errors. Below are a few excerpts from the Board report:

Page 5: "Although the failure was due to a systematic S/W design error..."

Page 6: "... presumably based on the view that ... not wise to make changes in S/W which worked well with Ariane 4".

Page 6: "... S/W is flexible ... thus encourages highly demanding requirements ... complex implementations difficult to assess."

Page 9 : "... it is evident that the limitations of the SRI S/W were not fully analysed..."

Page 12: "This loss of information was due to specification and design errors in the S/W of the inertial reference system."

Hence, the following recommendations from the Inquiry Board (excerpts):

"Prepare a test facility ... and perform complete, closed-loop, system testing. Complete simulations must take place before any mission."

"Review all flight S/W...".

"Set up a team that will prepare the procedure for qualifying S/W,..., and ascertain that specification, verification and testing of S/W are of a consistently high quality..."

Of course, improving the quality of the on-board S/W cannot do any harm. However, doing just that misses the real target.

What caught our attention in the first place was the following sentence (page 3 of the report):

"... In order to improve reliability, there is considerable redundancy at the equipment level. There are two SRIs operating in parallel with identical H/W and S/W."

This is really puzzling. Simple (i.e., degree 2) non diversified redundancy is the weakest kind of redundancy that can exist. How can this be deemed "considerable"? Furthermore, this is the only sentence of the report that addresses system design issues. Very likely, the Board quickly concluded that, given this "considerable" redundancy, system design issues deserved no additional attention.

There is some irony with the fact that a minor improvement w.r.t. redundancy, that is considering degree 2 diversified redundancy in the encoding of some variables, could have helped avoid the 501 failure.

More fundamentally, it is somewhat disturbing to observe that, once again, S/W is held to be "all what matters" a priori. We refer the reader back to section 3.5 for more on this topic.

This case study helps showing how proof-based SE can be useful a posteriori, to diagnose the causes of a failure. We present those system engineering faults which, in our view, are the real causes of the 501 failure. In the sequel, we use the term "module" rather than "computer" to refer to SRIs and OBCs. A module is a set comprising application S/W tasks, system level S/W tasks (e.g., algorithms), related data and variables, a computer. When a probability

of failure is estimated for a module, not just the computer (the H/W) should be taken into consideration.

5.4 Problem Capture Faults

Let $< X' >$ denote the actual specification established by the prime contractor and the industrial architect. Some of the necessary models (see section 3.1) were not specified in $< X' >$.

- Fault C_1 (models of shared persistent variables, SRIs): horizontal velocity, which is an external variable shared by the environment of the FCS and some of the SRI module tasks, had not been listed in $< m.X' >$. Neither N nor BH, which depend on horizontal velocity, had been listed either.

  Fault C_1 led to system dimensioning fault Dim_1.

- Fault C_2 (task models, SRIs): Condition "must be aborted after lift-off" had not been specified as an attribute for the alignment task in $< m.X' >$.

The alignment task was running after lift-off. The "exception condition" (BH overflow) was raised while running this task. It was later acknowledged that there is no need to keep this task running after lift-off in the case of Ariane 5. This has resulted into system design fault Des_3.

- Fault C_3 (failure models, SRIs): It was implicitly assumed that SRI modules would fail according to the *stop* model.

Firstly, assumptions must be explicitly specified. Implicit assumptions can be violated, which happened. Secondly, this assumption was flawed in the first place. SRI computers, not SRI modules, were taken into consideration. This led to system design faults Des_1 and Des_4.

- Fault C_4 (failure occurrence models, SRIs): It was implicitly assumed that 1 SRI module at most would fail during a flight.

Again, assumptions must be explicitly specified. This implicit assumption also was violated. SRI computers, not SRI modules, were considered. Specification $< m.X' >$ should have included the following statement: "up to f SRI modules can fail during a flight". Recall that the dimensioning of f is conditioned on fulfilling a proof obligation which, itself, depends on design decisions (see further). This led to system design fault Des_4 and to system dimensioning fault Dim_2.

Some of the properties were not properly specified in $< p.X' >$. Serializability is an example. Some internal variables shared among SRI module tasks had not been listed in $< m.X' >$. For these variables, appropriate invariants were not specified. (Note, though, that this was not a cause of the 501 failure).

Continuous correct SRI inertial data service provided to the OBCs is essential to a correct functioning of the launcher. Availability, in particular, is an essential

property. Availability was correctly specified, as follows: access to SRI inertial data service must be continuously ensured for the entire duration of a flight, with a probability at least equal to 1 - p, $p \approx 10^{-4}$. (It was implicitly assumed that reliability - the SRI inertial data service delivered is correct - was ensured, as "demonstrated" by the many successful flights of Ariane 4).

Had proof-based SE been applied, the prime contractor and the industrial architect would have developed a specification $< X >$ that would have been a correct translation of the real application problem specification $< A >$, i.e. free from faults C_1 to C_4. A first obvious conclusion is as follows: given that $< X' >$ is weaker than $< X >$ (in the $\supseteq$ sense), no FCS that (possibly provably correctly) solves $< X' >$ can be a correct solution to the real application problem considered.

5.5 System Design Faults

Redundancy of the SRIs is managed by the OBCs via a detection-and-recovery algorithm, denoted F_1 (passive redundancy). A correct detection-and-recovery algorithm must ensure that no OBC module behaves incorrectly because of failing SRI modules, which means that such an algorithm must be proved correct for the failure models that correctly reflect actual failure behaviors (see fault C_3). Such has not been the case. F_1 does not insulate the OBC modules from failed SRI modules behavior.

- Fault Des_1 (SRIs, OBCs): The detection-and-recovery algorithm run by the OBC modules to manage the redundant SRI modules is inconsistent with the actual failure behavior of SRI modules (or vice-versa).

This is an example of a fault that results from not having designed the FCS "in-the-large". Such faults can be avoided by fulfilling a design correctness proof obligation. That would have resulted into having system engineers select a consistent pair {failure model, algorithm F}. Consequently, the respective specifications of SRI and OBC modules would have been globally consistent. Let us give examples.

One possibility is to retain the current design, i.e. to choose $\{stop, F_1\}$. Choice $F = F_1$ is valid only if the *stop* assumption is shown to be actually instantiated vis-à-vis the OBC flight program (which commands the nozzle deflectors). It was believed that this assumption could not be enforced locally, at the SRIs, because of the 80% workload target. Therefore, the SRI *stop* assumption had to be enforced by the OBCs. That involves two requirements: (i) F_1 had to include a "filtering" mechanism that would discriminate between "flight data" and "exception condition reporting" messages issued by the SRI modules, (ii) two separate OBC tasks should have been specified, one in charge of executing the flight program, the other one in charge of handling exception conditions (error messages). Had this been done, *stop* behavior would have been correctly implemented vis-à-vis the OBC flight program. (This in fact should have been done - see fault Des_2).

One could also consider designs based on pairs {any failure model, masking algorithm F}. Recall that models stronger than *stop* have a higher coverage factor. For any such model, there are known solutions (e.g., algorithms F based on majority voting) for a synchronous computational model - which was retained for the FCS - as well as optimal feasibility conditions. In the presence of up to f failures, n modules are necessary and sufficient, with $n = 2f + 1$ (resp. $3f + 1$) if byzantine models are not (resp. are) considered. The latter is the design choice made for the US Space Shuttle on-board FCS. (Note that n is the behavioral function R introduced in section 3.2).

Given that other faults have also been made, doing the above would not have sufficed to avoid the 501 failure. Nevertheless, fault Des_1 is a latent fault that may lead to a future flight failure.

- Fault Des_2 (event models, task models, OBCs): Type of events "flight data" and type of events "error condition reporting" were not mapped onto two different OBC tasks.

Every event type 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 type} mapping must be specified. Had this been done, event type "overflow reporting" would not have been an input to the OBC flight program.

- Fault Des_3 (task models, SRIs): A task scheduling algorithm should have aborted the alignment task right after lift-off.

Given fault C_2, it was simply impossible for those system engineers in charge of designing/selecting the SRI task scheduler, as well as eligibility scheduling rules, to "guess" that the alignment task should have been aborted right after lift-off.

- Fault Des_4 (SRIs, OBCs): No proof was established that the probability of experiencing more than $f = 1$ SRI module failure during a flight is p at most.

Probabilistic computations that have been performed apply to SRI computers, rather than to SRI modules. This would have been revealed by attempting a proof that the stated availability property does hold. In doing so, system engineers would have run into the obligation of proving that the "up-to-f-out-of-n" assumption has a coverage factor greater than 1 - p. Which would have led them to prove that f cannot be equal to n, which is tantamount to proving that there is no common cause of failure for the SRI modules.

- Fault Des_5 (SRIs, OBCs): Given the system engineering approach followed, diversified redundancy should have been specified.

Indeed, making such deterministic assumptions as "up-to-f-out-of-n" (up-to-1-out-of-2 for the FCS), or any similar assumptions of probabilistic nature, does not make sense unless these assumptions are validated. Quite clearly, any capture, design, or dimensioning fault is a common cause of failure, which invalidates such assumptions. Whenever a non proof-based SE method is applied, it is safer - if not mandatory - to specify diversified redundancy, in order to eliminate potential common causes of failure, with a sufficiently "high" probability. This is well understood vis-à-vis implementation phases (H/W engineering, S/W engineering). This seems not to be so well acknowledged (yet) vis-à-vis system engineering phases.

System dimensioning fault Dim_1 (see below) would have been avoided had proof-based SE been resorted to. Conversely, under the existing approach, had diversified redundancy been specified, a large enough size could have been picked up (with a bit of luck) for the memory buffer used to implement BH on one of the two SRI modules.

Other space missions have failed for the same reason (Mars Observer is an example).

5.6 System Dimensioning Faults

No rigorous capture having been conducted, and system design faults having been made, some of the variables that should have appeared in $\{[S], [s]\}$, the generic FCS implementation specification, were missing (the case for BH), or had been valued a priori (the case for f). Given that no dimensioning correctness proof obligation was fulfilled, this was not detected at the time the actual FCS specifications were handed over to S/W engineers.

This resulted into assigning an incorrectly sized memory buffer to variable BH. This resulted also into an empirical dimensioning of the SRI logical modules group.

- Fault Dim_1: Range of values taken by variable BH was assumed to be in the $]-2^{15}, +2^{15}[$ interval.

Had proof-based SE been resorted to, fault C_1 would have been avoided. Consequently, $< m.x >$ would have included variables horizontal velocity, N and BH. In order to run the dimensioning phase, the prime contractor and/or the industrial architect would have then been forced to ask Ariane 5 program space flight engineers to decide upon some $V(< x >)$. That is how the correct value range of external variable horizontal velocity would have been explicitly specified, i.e. the range that is valid for Ariane 5, not for Ariane 4.

This would have translated automatically into a correct quantification of the value ranges for N and BH, variables that depend on that external variable. As a result, with a very simple oracle at hands, a correct dimensioning of the memory buffer assigned to BH - specified via $V([s])$ - would have been obtained.

It may be that, for Ariane 4, the issue of how to dimension the BH buffer was raised and solved. Problem is that the SE method applied for Ariane 5 led the prime contractor and the industrial architect to ignore that issue.

- Fault Dim_2: It was implicitly assumed that f's value would be 1 and that simple redundancy ($n = 2$) would suffice.

The intrinsic reliability of a SRI module (H/ W + S/W + data), and related confidence interval, can be derived from statistics computed over accumulated experimental data and simulation results, or via probabilistic modeling. Intrinsic reliability, combined with launch duration, is used to compute a value for f. Obviously, the smallest acceptable value for f is such that the probability of experiencing more than f SRI module failures is smaller than p.

It appears that system engineers took SRI computers (i.e., H/W) only into consideration. Had SRI modules been considered rather, related confidence intervals on modules reliability could have led to choose $f = 2$ (for example).

Given that other faults have been made, picking up a value of f (resp. n) greater than 1 (resp. 2) would not have helped avoid the 501 failure. Nevertheless, this is a latent fault that may lead to a future flight failure.

The dual architecture of the FCS reflects the following widespread SE practice for many embedded systems : design for a "single-copy" architecture (1 SRI, 1 OBC in the FCS case) and duplicate.

From a more general perspective, if we compare the costs involved with applying a proof-based SE method and implementing a rigorously designed and dimensioned embedded system on the one hand, costs incurred with operational failures - such as that of the 501 failure - on the other hand, it seems there are good reasons to consider switching to proof-based SE.

5.7 Summary

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 problem. Faults have also been made in the course of designing and dimensioning the Ariane 5 FCS. These faults have been activated during launch, which has caused errors at the FCS modules level. Neither algorithmic nor architectural solutions could cope with such errors. Flight 501 failure ensued.

Flight 501 also is an example showing that integration testing phases - as conducted under current SE practice - are not satisfactory, for they are unavoidably incomplete. Flight 501 can be viewed as a (costly) continuation of an incomplete integration testing phase.

This case study also helps showing that S/W reuse - a very sound objective - has far reaching implications, which promote reliance on proof-based SE. Reuse of S/W modules - those which work fine w.r.t. the Ariane 4 application problem - is possible provided that those conditions under which they are to operate are kept unchanged. Which is not the case when considering the Ariane 5 application problem, or its quantification to the very least. As shown above, proof-based SE, which permits to address these issues rigorously, favors S/W reuse.

Let us now go back to the conclusions of the Inquiry Board and ask ourselves the following questions:

- Had all the modules as specified by system engineers been implemented in H/W, would Flight 501 have failed the way it did?
- Had the implementation - in S/W and/or in H/W - of every module as specified by system engineers been proved correct, would Flight 501 have failed the way it did?

Obviously, answers are "yes" in both cases. Specifications of the S/W modules as established by system engineers were incomplete/incorrect w.r.t. real problem $< X >$. There is no "S/W error" involved with that. Why should the concept of horizontal velocity be viewed as a S/W engineering concept rather than a H/W engineering concept? (It is neither). What if the conversion procedure which computes values of BH had been implemented in H/W? Flight 501 would have failed as well. However, the Inquiry Board would have then diagnosed a "H/W design error". Clearly, causes of the failure belong neither to the "S/W world" nor to the "H/W world". Issues at stake are system engineering issues.

The S/W implemented N-to-BH conversion procedure is absolutely correct. For any input value N, it computes a correct value BH. Not enough bits (15 instead of 18) were allocated for storing these correct values. Calling a faulty dimensioning of a memory buffer a "S/W error" is as misleading as calling the choice of too slow a processor a "S/W error".

Confusion between S/W engineering and system engineering should come to an end.

As for the recommendations of the Inquiry Board that system testing and system simulation must be complete, we refer the reader to sections 3.4 and 3.5. As we know, completeness - without resorting to proof-based SE - is beyond reachability. Such recommendations are not very useful for the simple reason that they cannot be implemented, if one believes that S/W is the only area of concern.

6 Conclusions

The principles presented in this paper are believed to be useful in helping set the foundations of proof-based system engineering for computer-based systems. These principles have been illustrated with a presentation of a proof-based SE method which has been applied to various problems, in cooperation with a number of companies and agencies. This has permitted to validate and to refine these principles.

The essential feature of the methodological revolution embodied in proof-based system engineering is that it has the effect of converting system engineering into a bona fide scientific discipline. Every technical or methodological field goes through this transition. When it does, advances occur at an unimaginable pace.

Times of changes are upon us when an area goes from trial-and-error to prediction based on knowledge of the inner mechanisms and principles at work. We are now on that threshold in the field of system engineering for computer-based systems.

References

1. Bernstein, P. A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Pub., ISBN 0-201-10715-5 (1987) 370 p.
2. Gaubert S., Max Plus: Methods and Applications of $(max, +)$ Linear Algebra, INRIA Research Report 3088 (Jan. 1997) 24 p.
3. Hadzilacos, V., Toueg, S.: A Modular Approach to Fault-Tolerant Broadcasts and Related Problems, Technical Report, TR 94-1425, Cornell University (May 1994).
4. Hermant, J.F., Le Lann, G.: A Protocol and Correctness Proofs for Real-Time High-Performance Broadcast Networks, 18th IEEE Intl. Conference on Distributed Computing Systems, Amsterdam, NL (May 1998) 10 p.
5. Hermant, J.F., Leboucher, L., Rivierre, N.: Real-Time Fixed and Dynamic Priority Driven Scheduling Algorithms: Theory and Experience. INRIA Research Report 3081 (Dec. 1996) 139 p.
6. Inquiry Board Report: Ariane 5 - Flight 501 Failure, Paris (19 July 1996) 18 p. [http://www.inria.fr/actualites-fra.html].
7. Joseph, M., et al.: Real-Time Systems - Specification, Verification and Analysis. Prentice Hall UK Pub., ISBN 0-13-455297-0 (1996) 278 p.
8. Knightly E.W., Zhang Hui: D-BIND: An Accurate Traffic Model for Providing QoS Guarantees to VBR Traffic. IEEE/ACM Transactions on Networking, vol. 5, n^{o}2 (April 1997) 219 - 231.
9. Koren, G., Shasha, D.: D-Over: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems, INRIA Technical Report 138 (Feb. 1992) 45 p.
10. Kuhn, D.R.: Sources of Failure in the Public Switched Telephone Network, IEEE Computer (April 1997) 31 - 36.
11. Le Lann, G.: Certifiable Critical Complex Computing Systems. 13th IFIP World Computer Congress, Duncan K. and Krueger K. Eds, Elsevier Science Pub., vol. 3, Hamburg, D (Aug. 1994) 287 - 294.
12. Le Lann, G.: Proof-Based System Engineering for Computing Systems, ESA-INCOSE Conference on Systems Engineering, Noordwijk, NL (Nov. 1997) IEE/ESA Pub., vol. WPP-130, 5a.4.1 - 5a.4.8.
13. Le Lann, G.: An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective. IEEE Intl. Conference on the Engineering of Computer-Based Systems, Monterey, USA (March 1997) 339 - 346.
14. Leveson, N.G., Turner, C.: An Investigation of the Therac-25 Accidents, IEEE Computer (July 1993) 18 - 41.
15. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Pub., ISBN 1-55860-348-4 (1996) 872 p.
16. Neumann, P.G.: Computer Related Risks, Addison-Wesley Pub., ISBN 0-201-55805-X (1995) 367 p.
17. Powell, D.: Failure Mode Assumptions and Assumption Coverage, 22nd IEEE Intl. Symposium on Fault-Tolerant Computing, Boston, USA (July 1992) 386-395.

This article was processed using the LaTeX macro package with LLNCS style