PROOF-BASED SYSTEM ENGINEERING FOR COMPUTING SYSTEMS

Gérard Le Lann
INRIA, Projet REFLECS, BP 105
F-78153 Le Chesnay Cedex, France
E-mail: Gerard.Le_Lann@inria.fr
Fax: +33.139.63.58.92

ABSTRACT

We introduce basic principles that underlie a scientific approach to system engineering for computing systems. This approach is based upon fulfilling proof obligations, notably establishing proofs that system design and system dimensioning decisions are correct, before embarking on the implementation or the fielding of a computing system. A proof-based system engineering method which has been applied to diverse projects involving real-time distributed fault-tolerant computing systems is also presented. Lessons learned are reported. Furthermore, we elaborate on why proof-based system engineering helps in slashing project delays and budgets.

1. INTRODUCTION

A study conducted by the Standish Group [Ref. 1] confirms that a very large percentage of projects involving computing technology are unsuccessful. Most projects are significantly delayed, or are cancelled, or entail costs much higher than anticipated, or result into operational failures. A first question is “Why is it so?”. A second question is “What is being done to successfully address the customers defined ABC (Asap, Better, Cheaper) challenge?”.

Over the last 10 years, INRIA’s Projet REFLECS has contributed to a number of audits, responses to invitations-to-tender, design work for real-time distributed fault-tolerant computing systems. This is how we came to realize that system engineering - denoted SE - 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 [Ref. 2]. Indeed - and contrary to widespread belief in certain circles - the dominant cause of project setbacks and operational failures is not poor software (S/W) engineering practice. This view is being shared by a growing number of experts, and supported by detailed and extensive studies, such as, e.g. [Refs. 3, 4]. This is also the rationale that underlies the INCOSE initiative (see [Ref. 5] for an introduction).

Hence, the answer to the first question is “Poor SE practice”. The answer to the second question is “Not much”. This may sound a bit harsh. Let’s face it: if we are to *successfully* meet the ABC challenge, we must admit that time has come for drastic changes in SE methods, processes and tools directed at computing systems. This is very much comparable to the “10X changes” or “strategic inflection points” identified in the IC industry [Ref. 6].

As is the case with every mature engineering discipline, SE for computing systems has to build upon sciences. For instance, let us consider the body of knowledge that has been accumulated over the last 25 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 industry have solutions readily available. To be more specific, there are many architectural and algorithmic problems that have provably optimal or correct solutions, notably in the areas of real-time computing, distributed computing, and fault-tolerant computing. However, only a small subset of these solutions have so far translated into commercial, off-the-shelf (COTS) products or operational computing systems.

We believe that proof-based SE - introduced in section 2 - is the privileged vehicle to successfully meet the ABC challenge. This belief is supported by a number of experiments and feasibility studies, which are reported in section 3. In section 4, we explain why proof-based SE circumvents those weaknesses that plague current SE methods and processes.

2. WHAT IS PROOF-BASED SYSTEM ENGINEERING

2.1 Introduction

With proof-based SE, one seeks to solve system engineering problems arising with computing 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). Why would it be different when it comes to computing 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 users/customers requirements and assumptions (e.g., an invitation-to-tender), to produce a global and implementable specification of a computing 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 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".

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 at INRIA in 1993 [Ref. 7] has led to the inception of a SE method named TRDF - which stands for Temps Réel, Traitement Distribué, Tolérance aux Fautes (T is a common factor). That name was retained for the reason that those application problems of interest to us raise combined real-time (R), distribution (D), and fault-tolerance (F) 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 will proceed with presenting proof-based SE as instantiated by the TRDF method.

2.2 The three phases

Notation <N, n> is used to refer to a specification of a computer science problem or solution, N being a set of properties, n being a set of assumptions. The term "specification" is used to refer to any complete and unambiguous statement - in some human language, in some formalized notation, in some formal language. Notation [N, n] is used to refer to a specification of a generic application problem. Such a specification is usually established by a customer, in some human language.

The three phases of a project life cycle which fall within the scope of proof-based SE, and which precede phases covered by other engineering disciplines (e.g., S/W engineering), are now briefly introduced. The whole set of notations needed to describe these phases is summarized in figure 1.

[AP, ap]	≡	specification of the invariant (i.e., generic) subset of an application problem
<CP, cp>	≡	specification of the generic computer science problem that matches [AP, ap]
<Φ, φ>	≡	specification of a particular quantification of <CP, cp>
<D, d>	≡	modular specification of generic S that satisfies <CP, cp>
<Ξ>	≡	specification of a dimensioning Oracle (obtained from design correctness proofs) for generic S
<V, v>	≡	modular specification of a physical dimensioning (of generic S) that satisfies <Φ, φ>

$$\{\text{description of an Application problem}\} \Rightarrow [AP, ap] \;\Big|\; \begin{array}{l} \Rightarrow \; < CP, cp > \\ \Rightarrow \; < \Phi, \varphi > \end{array}$$

Capture

$$< CP, cp > \;\Big|\; \begin{array}{l} \Rightarrow \; < D, d > \\ \Rightarrow \; < \Xi > \end{array}$$

Design

$$< \Phi, \varphi > \;\Rightarrow\; \Xi \;\Rightarrow\; < V, v >$$

Dimensioning

Figure 1. *Organization of project phases with proof-based SE*

2.2.1 The capture phase

This phase has an application problem description as an input. Such a description, provided by a customer, usually is incomplete and/or ambiguous. A capture phase is concerned with, (i) the translation of an application problem description into [AP, ap], which specifies the generic part of the application problem under consideration, and, (ii) the translation of [AP, ap] into <CP, cp>, a specification of a generic problem in computer science, as well as the construction of a specification denoted <Φ, φ>. A capture phase is jointly conducted by a customer and a prime contractor.

A generic problem is an invariant for the entire duration of a project. What is not included in [AP, ap] (or in <CP, cp>) is considered to be modifiable or to be defined later, which is specified via <Φ, φ>. Most often, [AP, ap], hence <CP, cp>,

contains unvalued variables. Via <Φ, φ>, such variables are assigned specific values, which reflect a particular instantiation of the application problem under consideration.

<CP> (resp. <cp>) specifies properties (resp. models) that have well defined semantics, such as those commonly considered in computer science [Refs. 8, 9, 10]. One important feature is that hierarchies or partial orders can be defined over these sets of models and properties, which leads to the concept of dominance, denoted ⊒. Dominance is related to the concept of an adversary in game theory. X ⊒ Y means that X (a model, a property) is stronger than or equal to Y (a model, a property). For example, the byzantine failure model is stronger than the crash failure model. Similarly, the bounded jitters (timeliness) property is stronger than the latest termination deadlines (timeliness) property.

2.2.2 The system design phase

This phase entirely is under the responsibility of a prime contractor. A design phase has a problem <CP, cp> as an input. It covers all the design stages needed to arrive at a modular specification of a generic solution (generic S), the completion of each design stage being conditioned on fulfilling a correctness proof obligation. A design stage is conducted by exploiting state-of-the-art in computing system architectures and algorithms (in addition to models and properties), as well as by applying appropriate proof techniques, which techniques depend on the types of problems under consideration. At any given design stage, a system design decision Δ consists in specifying a set of properties - denoted *prop*, a set of models - denoted *mod*, and an architectural and algorithmic solution - denoted A. A design proof consists in establishing that *prop* holds with A, under *mod*. For example, one proves that some algorithm EX enforces *prop* ≡ mutual exclusion under *mod* ≡ non-shared memory distributed architecture. Let us assume that deadlocks can occur with EX (which does not contradict *prop*).

The first stage of a design phase results into the breaking of the initial (possibly complex) generic problem <CP, cp> into a number of independent and simpler generic subproblems $<CP, cp>_i$. Design correctness proofs guarantee 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.

Even if proven correct, a design decision (a solution) may not suit a given problem. Hence, the concept of a system design correctness proof obligation, denoted K(Δ), which is defined as follows:

given problem $<CP, cp>_i$, a system design solution Δ(*prop*, A, *mod*) is correct iff the following conditions are satisfied:

$$<prop(A)> \supseteq <CP>_i$$
$$<mod> \supseteq <cp>_i$$

Pursuing with the example above, assume that $<CP>_i \equiv$ mutual exclusion + liveness (and that $<cp>_i \equiv mod$). The first condition of obligation K(Δ) cannot be satisfied with a design based on EX.

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 <CP, cp>. This modular specification is denoted <D, d>.

As explained above, <D> specifies, in particular, all the generic architectural solutions accumulated during a design phase. These generic solutions inevitably involve unvalued variables (e.g., a boolean matrix reflecting how application S/W modules can be mapped onto the future modules of S), given that <CP, cp> contains unvalued variables. Let <Configurator> be the specification that contains - in particular - the whole set of unvalued variables related to those generic architectural decisions stated in <D>. Establishing design correctness proofs yields generic feasibility conditions. Such conditions are a set of constraints that, for some given pair {problem, architectural and algorithmic solutions}, bind properties and models altogether. Let <feasibility Oracle> be the specification that contains those generic feasibility conditions corresponding to generic solution <D, d>. Let <Ξ> be the union of <Configurator> and <feasibility Oracle>. <Ξ> specifies a tool Ξ called a dimensioning Oracle.

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 computing systems.

2.2.3 The system dimensioning phase

This phase includes a single stage. It is conducted each time <Φ, φ> is "frozen" or filled in, by running Ξ with <Φ, φ> as an input. If the feasibility Oracle declares that problem <CP, cp>, quantified as per <Φ, φ>, is feasible, Ξ produces a specification <V, v>, which is a quantification of <D, d>, i.e. a physical dimensioning of generic S. <V, v> gives valuations of those implementation variables that appear in <D, d> (e.g., sizes of memory buffers, sizes of data structures, processors speeds, databuses throughputs, number of databuses, processors redundancy degrees, total number of processors, etc.). It may be that some of the valuations are chosen beforehand. This is indeed the case whenever COTS products are selected a priori.

That a problem quantification <Φ, φ> is feasible with some design Δ means that a number of system dimensioning correctness proof obligations are fulfilled. Such an obligation, denoted K(∇), is defined as follows:

given system design Δ(*prop*, A, *mod*), a system dimensioning ∇ is correct iff the following conditions are satisfied:

$$\nabla\langle prop(A)\rangle \supseteq \langle\Phi\rangle$$

$$\nabla\langle mod\rangle \supseteq \langle\varphi\rangle$$

For example, every task has its computed upper bound on response times (*prop*(A)) smaller than or equal to a quantified termination deadline (<Φ>), under computed load conditions (*mod*) greater than or equal to quantified conditions (<φ>).

Proof-based SE also addresses those changes that may impact a computing system after it has been fielded. Phases concerned with handling modifications in customer 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 above.

Pair {<D, d>, <V, v>} is a modular implementation specification of a computing system S that provably solves initial problem {<CP, cp>, <Φ, φ>}. Modules of {<D, d>, <V, v>} are contracts between a prime contractor and those co/sub-contractors in charge of implementing S. A dimensioning Oracle may be exploited by a prime contractor or by a customer.

Specification {<D, d>, <V, v>} 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. 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 computing systems. For example - as amply demonstrated by many failed projects and/or operational failures - a faulty capture of application-centric services results into deficient specifications of application S/W modules. Specifications handed over to S/W engineers must first be proven correct (w.r.t. some customer defined application problem). S/W engineering necessarily follows system engineering in a project life cycle.

3. LESSONS LEARNED WITH PROOF-BASED SYSTEM ENGINEERING

In this section, we give examples of projects where the TRDF method has been applied. In these projects, INRIA took on the role of a prime contractor. For the sake of conciseness, presentations of projects will simply consist in summarizing lessons learned. Work conducted during these projects has not been published yet. Reports, in French, are available upon request. The TRDF method is currently being applied to Air Traffic Control and critical Transactional applications.

3.1 Project 1

Customers: 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 <D, d> and <Ξ>. Essential 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, i.e. Δ_1, was to be undertaken. It turns out that specification [AP, ap] we had to consider is an invariant application problem for such drastically diverse areas as Business (Stock Markets, Currency Trading), Defense (C^3I, Space), Air Traffic Control, Nuclear Power Plants.

Sketch of problem properties <CP>:

- Safety (distribution):
  - Exactly-once semantics property for tasks. No task roll-backs.
  - Serializability property: every possible concurrent interleaved execution of tasks is equivalent to a sequential execution; invariants defined over shared updatable persistent variables are never violated.
- Timeliness (real-time): Task timeliness constraints = strict latest termination deadlines.
- Dependability (fault-tolerance):
  - Very high availability for network services.
  - Finite bounded latency for detecting processor failures.
- Complexity C(Ξ): Linear in the number of tasks.

Problem <CP, cp> derived from [AP, ap] is NP-hard. Solution A, specified in delivered <D>, includes a distributed clients-servers architecture and a distributed 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 Ξ:
precomputed schedule templates (not precomputed schedules) and particular constructs on graphs helped reduce C(Ξ) to what was specified.

Examples of techniques and results that were used are optimal scheduling algorithms [Ref. 11], analytical calculus, time-based agreement algorithms, matrix calculus in (max,+) algebra, dependability proofs and related feasibility conditions.

Lessons learned
The genericity of the TRDF method has been validated. It has also 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" under the form of {[AP, ap], <CP, cp>, <Φ, φ>} is instrumental in clarifying responsibilities. At one point, one of the customers felt that solution A "had a problem". It did not take long to agree that A was correct vis-à-vis the "contract" specified. What this customer had in mind was a variation of [AP, ap]. It turned out that the variation could be accommodated by existing design solution A, by restricting potential choices w.r.t. the schedule templates. However, the feasibility Oracle had to be partially revisited.

What has also been clearly verified is how fast a dimensioning Oracle can be compared to an event-driven simulation tool. For the task sets considered, run times never exceeded one hour with dimensioning Oracle Ξ, whereas run times were in the order of one day for the event-driven simulator used by one of the customers.

3.2 Project 2

Customer: 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 customer was delivered specifications [AP, ap] and <CP, cp>. 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 <*prop*, A, *mod*>. 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 <*mod*> and solution A, which led us to specification <*prop*>. We could then establish that none of the conditions of correctness proof obligation K(Δ) was fulfilled. Consequently, the COTS system considered was not suitable.

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

3.3 Project 3

Customers: French Ministry of Research and Dassault Aviation (Projet Génie). We had to identify what were the dependability properties effectively enforced with a simple embedded system considered for commanding and controlling essential actuators in future planes. Our work in this project was a bit similar to that of project 2. The TRDF method allowed to show that despite its 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.

Besides this work, different teams of S/W engineers and scientists applied formal and proof-based methods to develop those S/W modules needed to command and control the actuators. By the end of the project, proofs of S/W correctness had been established.

Lessons learned
What was revealed by applying the TRDF method were the exact "gaps" existing between the 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 operational environment. S/W correctness proofs were developed assuming models much weaker (in the ⊇ sense) than the correct ones.

This is an illustration of the fact that it is not a good idea to apply formal S/W methods without applying proof-based system engineering methods as well. Models that can currently be accommodated with formal S/W methods are ideal representations of physical reality and real computing systems. Hence the following choices: (i) either ignore these 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 computing systems those “simple” models that can be accommodated with existing formal S/W engineering methods.

3.4 Project 4

Customers: None. On 4 June 1996, the maiden flight of the European Ariane 5 satellite launcher ended in a failure. Conclusions of the Inquiry Board were issued on 19 July 1996, as a public report. According to the Inquiry Board, causes of the failure of flight 501 are S/W specification and S/W design errors. 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 (computers) 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 which addresses system design issues. Very likely, the Board quickly concluded that, given this “considerable” redundancy, system design issues deserved no additional attention.

As with project 2, we applied the TRDF method in order to analyze the Ariane 5 Flight Control System (FCS) as well as the failure scenario. Our conclusions deviate significantly from the Inquiry Board findings. We diagnosed 4 capture faults, 5 design faults and 2 dimensioning faults [Refs. 12, 13]. These system engineering faults are the real causes of the 501 failure. S/W errors diagnosed by the Inquiry Board simply are manifestations of these faults. As with other studies (e.g., [Ref. 3]), we hope that our analysis can help understand that it is inappropriate - possibly dangerous - to systematically “blame” the S/W engineers involved.

Granted, reusability of S/W modules is a very good principle ... provided that those conditions under which a seemingly correct S/W module is going to operate are fully and unambiguously specified. In particular, this translates into specifying ranges of values for variables. This is a S/W engineering issue for variables that are strictly internal to a S/W module. This is a system engineering issue for variables that are shared between a S/W (H/W as well) module and its physical environment. Why should the concept of horizontal velocity of a launcher be viewed as a S/W engineering issue? Had the conversion program that computes BH been implemented in H/W, flight 501 would have failed as well. Would the Inquiry Board have then “blamed” H/W engineers? 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”.

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. Lack of proofs during design stages results into integration tests whose complexity is exponential in the total number of states that can be exhibited by the FCS modules. Lack of proofs translates into lack of time (and budgets, as well). Indeed, flight 501 can be viewed as a (costly) continuation of an incomplete integration testing phase. Which explains why the recommendations of the Inquiry Board that system testing and system simulation must be *complete* are not very useful. As we know, completeness - without proof-based SE - is beyond reachability. Such recommendations simply cannot be implemented, if one believes that “S/W is all what matters”.

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.

4.1 Technical advantages

- Design reusability. Whenever the capture of a new application problem yields an invariant [AP, ap] which has been translated into <CP, cp> and provably solved in the past with generic design <D, d>, that design comes for free for this new application problem. (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 customer to select at will any combination of application S/W modules, some quantification of [AP, ap], and to

be delivered a correctly dimensioned computing system S, without having to incur any of those delays and costs that are induced by the need to re-design S.

- Elimination of 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 life cycle.

- Rigorous assessment of COTS technology, via correctness proof obligations K(Δ) and K(∇).

- Simplification of final integration testing. Verification that a set of concatenated modules "behaves correctly" runs into combinatorial problems. Most often, verification is done via testing, within imposed bounded time, which means that testing is necessarily incomplete, even with computing 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 proven beforehand that the union of modules is a global solution. If it would be the case that every specification module $\{<D, d>, <V, v>\}_i$ 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 down the complexity of system integration testing to acceptable figures, such as, e.g., pseudo-linear complexity.

4.2 Strategic advantages

It is common practice to assign different pieces of a project to different contractors. It is possible for a customer or a prime contractor (say X) 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 with a precise specification $<CP, cp>_i$. X may ask contractors to show proofs of design and dimensioning correctness. In case a problem arises, responsibilities are clearly identifiable. Furthermore, note that X only has complete knowledge of the overall technical decisions. Therefore, X may subcontract work, even to its competitors, without giving away its technical know-how.

4.3 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 seeing that the problems are due to incorrect dam blueprints.

Whenever a computing 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 customers and/or contractors. Planned or delivered computing 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 computing 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. Which translates into showing that a set of invariants (C) is always satisfied. This is known as the serializability problem, that has many solutions readily available [Ref. 8].

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 (C) is not violated. Let us make some rough calculation. 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 asynchronous computations - the correct paradigm when concerned with failures and critical applications - and which are based on exhaustive searches of state spaces, are doomed to failure, even with modestly complex systems. Furthermore, their usage is hardly justifiable, given that there exists an impressive gamut of concurrency control algorithms. Companion proofs establish that a computing system equipped with any of these algorithms guarantees that (C) 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.

5. CONCLUSIONS

The principles presented in this paper are believed to be useful in helping set the foundations of proof-based system engineering for computing systems. These principles have been illustrated with a presentation of how a proof-based SE method developed within INRIA's Projet REFLECS has been applied to various real world problems, in cooperation with customers and contractors. This has permitted to validate and to refine these principles. An extensive presentation of proof-based SE for computing systems and computer-based systems can be found in [Ref. 14].

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 computing systems.

REFERENCES

1. The Standish Group, The Scope of Software Development Project Failures, 1995.

2. White S. et al., Engineering of Computer-Based Systems: Current Status and Technical Activities, IEEE Computer, June 1995, 100-101.

3. Leveson N. G., Turner C., An investigation of the Therac-25 accidents, IEEE Computer, July 1993, 18-41.

4. Kuhn D. R., Sources of Failure in the Public Switched Telephone Network, IEEE Computer, April 1997, 31-36.

5. IEEE Transactions on Aerospace and Electronic Systems, vol. 33 (2), April 1997, 577-733.

6. Grove A., Only the Paranoids Survive, Currency/Doubleday Pub., 1996, 210 p.

7. Le Lann G., Certifiable Critical Complex Computing Systems, 13th IFIP World Computer Congress, Duncan K. and Krueger K. Eds, Elsevier Science Pub., vol. 3, 1994, 287 - 294.

8. 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.

9. Joseph, M., et al., Real-Time Systems - Specification, Verification and Analysis, Prentice Hall UK Pub., ISBN 0-13-455297-0, 1996, 278 p.

10. Lynch, N.A., Distributed Algorithms, Morgan Kaufmann Pub., ISBN 1-55860-348-4, 1996, 872 p.

11. 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.

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

13. Le Lann, G., An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective, IEEE Conference on Engineering of Computer-Based Systems, Monterey, CA, March 1997, 339-346.

14. Le Lann, G., Proof-Based System Engineering and Embedded Systems, School on Embedded Systems, Veldhoven (NL), Nov. 1996, Lecture Notes in Computer Science, Springer Verlag Pub., to appear, 40 p.