A Methodology for Designing and Dimensioning Critical Complex Computing Systems

Gérard Le Lann
INRIA, Projet REFLECS, France
Gerard.Le_Lann@inria.fr

Abstract

It is widely recognized that real-time, fault-tolerant and distributed computing technologies play a key role in the deployment of many current and future (civilian or Defense) critical and complex applications. Computing systems needed to support such applications are referred to as C^3 systems. Lack of a clear identification of those issues involved with designing and dimensioning C^3 systems can only lead to failures, as recently demonstrated by a number of sizeable projects that have been aborted or suspended in Europe and in the USA, in various application domains.

This paper describes a Systems Engineering methodology that, given some specification <P, p> of a particular Systems Engineering problem, permits to develop a specification <S, s> of a C^3 system such that <S, s> provably satisfies <P, p>. It is explicitly assumed that <P, p> includes arbitrarily stringent timeliness requirements, arbitrary distribution requirements as well as arbitrarily stringent dependability requirements. Moving from <P, p> to <S, s> involves some number of design stages and one final dimensioning stage. It is shown how to verify whether every single design decision satisfies the logical part of <P, p> as well as whether a dimensioning decision satisfies the physical part of <P, p>.

This methodology is fully orthogonal to formal specification methods or formal software engineering methods currently in use. It does not rest on any particular programming language either. Too often, system design and/or system dimensioning stages are conducted in ad-hoc ways, or even confused with implementation or software development. We believe this to be the main reason why so many complex systems fail to operate correctly or are abandoned, contrary to widespread belief that software faults are the primary culprit.

The formal aspects of the methodology are related to demonstrating that specific safety, timeliness and dependability properties are enforced by a given design. Such demonstrations lead to provably correct generic (i.e. reusable) designs. Proofs that play a prominent part in this methodology are called timeliness proofs («hard» real-time properties), serializability proofs (safety properties) and dependability proofs (availability properties). Examples of some techniques used to establish such proofs in the presence of incomplete advance knowledge of the future are given.

1. Introduction

The purpose of C^3 **systems** is to match those requirements found in Invitations-To-Tender (ITTs) issued by Governmental Bodies or national/international companies or consortia operating in various business areas (e.g., banking/ finance, Defense, telecommunications, air traffic control, on-line reservation). Issuers of ITTs will be referred to as **clients** in the sequel.

There is growing evidence that those **system providers** in charge of developing C^3 systems have faced or are facing serious difficulties, which manifest as deadlines being missed repeatedly as well as by seemingly unbounded inflated budgets. As a result, many developments are suspended or even canceled by the clients.

Recently, this has been the case for the new US Air Traffic Control project, the UK Taurus project (computerization of the London Stock Exchange, 425 million £ spent, project abandoned), the US Orbital Station On-Board Data Management system (500 million $ spent, project abandoned), the US on-line reservation services Confirm project (125 million $ spent, project cancelled), to name a few. In other instances (e.g., Socrate, the French Railways Seat Reservation system or Relit, the Paris Stock Exchange system), C^3 systems are put into operation but fail to operate correctly too often (crashes or abnormally large response times).

As it seems reasonable to assume that system providers do their best to design and build such C^3 systems, it must be that something quite essential is overlooked or addressed empirically under current industrial/engineering practice. This «something» is **Systems Engineering**, whose role and nature are discussed in section 2. In section 3, we introduce a methodology for the engineering of C^3 systems, which is based on **proof obligations** applied to the **design** and the **dimensioning** stages of such systems. Examples of useful proof techniques are given in section 4.

This methodology results from a 3-year long research initiative undertaken by the REFLECS group at INRIA (Rocquencourt, France). The methodology currently is being applied within the framework of two projects (one civilian application and one Defense application), each involving a major French system provider. The methodology also is being considered by European clients and system providers.

0-8186-7355-9/96 $05.00 © 1996 IEEE

2. Systems engineering and C^3 systems

2.1. Systems Engineering

The meaning of notation <G, g> used throughout the paper is as follows: under assumptions g, properties G should hold. Consider the generic example of an ITT for a critical and complex application, as shown on figures 1 and 2, where Ω is a description of the application requirements and ω a description of those assumptions under which Ω should hold. In particular, ω is a description of the future system environment, denoted E.

Only the logical part of an ITT, denoted $<\Omega(l), \omega(l)>$, is needed to start the design stages. The physical part, denoted $<\Omega(p), \omega(p)>$, may be initially undefined. Furthermore, this part is prone to numerous changes, during the design stages as well as after a provably correct design has been established.

$\Omega(l)$: N, the number of services (set Σ) provided by the application S/W, is bounded but unknown. Every service $s \in \Sigma$ is instantiated as a S/W component. The design of such components is independent from any OS or H/W technology. These components share on-line updatable persistent data structures. A set I of invariants (consistency constraints) is defined over the data structures.
Every possible combination of components must satisfy I. That is, S/W components must execute correctly despite arbitrary concurrency. Some services are mutually exclusive.
For every possible run of any combination of components, any request for service $s \in \Sigma$ must be completed in d_s time units at most.
Some services are critical. Unavailability of critical services (u) should not exceed $10^{-\alpha}$, α in the order of 9 (at most).

$\omega(l)$:Distribution of shared data structures is arbitrary. Distribution of every S/W component is arbitrary. S/W components are modeled as arbitrary directed finite graphs. Every single S/W component satisfies I when run alone.
A maximum execution time is known for every S/W component when run alone, denoted x_s for service $s \in \Sigma$. Arrival laws of service requests are arbitrary. For every service s, an upper bound on arrivals density is given as the ratio a_s/w_s, where w_s is a (sliding) time window and a_s is the highest number of requests that can be found in any time window of size w_s. Up to p processor crash failures over D time units can be caused by the environment.

Figure 1. A generic example of an ITT, logical part

$\Omega(p)$: N = 1200 ; α = 8. Variables d_i's in milliseconds: $d_1 = 12$ $d_2 = 850$ ... $d_N = 110$.

$\omega(p)$: p = 5 ; D = 14 hours. Variables x_i's in milliseconds: $x_1 = 3$ $x_2 = 160$ ... $x_N = 4$. Variables w_i's in seconds: $a_1/w_1 = 4/0.3$ $a_2/w_2 = 5/15$ ... $a_N/w_N = 76/9.5$.

Figure 2. A generic example of an ITT, physical part

It is the responsibility of a designer to translate $<\Omega(l), \omega(l)>$ into a non ambiguous **specification of a generic Systems Engineering problem, denoted $<\Lambda, \lambda>$**. No formal method being known yet to do this in a formal way, this translation is necessarily interpretative and iterative, until both the client and the designer agree that $<\Lambda, \lambda>$ is indeed a "correct" capture of $<\Omega(l), \omega(l)>$. Beyond this capture stage, correctness proofs are mandatory.

The final goal pursued by a designer is to produce a specification, denoted <D, d>, of a system, denoted SYS, such that <D, d> provably satisfies $<\Lambda, \lambda>$. **Specification <D, d> is a generic solution to generic problem $<\Lambda, \lambda>$**, much like in Mathematics, where the game consists in demonstrating theorems (G) under some axiomatics (g). That is, design correctness proofs hold true for every possible valuation of those variables appearing in $<\Lambda, \lambda>$.

Along with <D, d>, a designer should specify a tool whose entries are $<\Lambda, \lambda>$ and any of the many (future) descriptions $<\Omega(p), \omega(p)>$ that can be contemplated by a client. This tool translates $<\Omega(p), \omega(p)>$ into a specification $<\Phi, \varphi>$, which is a particular valuation of $<\Lambda, \lambda>$. Every possible pair $<<\Lambda, \lambda>, <\Phi, \varphi>>$, denoted <P, p>, specifies a particular valuation of the generic problem considered. Such a tool includes a feasibility Oracle, whose role is to return a verdict along with some data structures. In case the verdict is negative, reasons are given via these data structures. Whenever the verdict is positive, i.e. whenever the valued Systems Engineering problem considered has a solution, the Oracle produces <V, v>, which is a specification of how to dimension the implementation of SYS.

Proofs that verdicts and dimensioning decisions are correct must be given along with the tool.

Hence, whenever a valued Systems Engineering problem is declared feasible, the outcome is a specification <S, s> = <<D, d>, <V, v>> which guarantees that system SYS will always "win against" environment/"adversary" E. Issues raised with how to correctly implement <S, s>, that are addressed by, e.g., S/W Engineering and Electrical/Optical Engineering methods, fall outside the scope of the methodology presented in section 3.

It is important to understand that programming issues need not/should not be addressed before a provably correct specification <S, s> is obtained. A simple way of understanding the difference which exists between Systems

Engineering and Software Engineering is by observing that a C^3 system whose software is fully correct (i.e. absolutely fault-free) can only fail if what is (perfectly) coded are incorrect or inappropriate algorithms and architectures, possibly incorrectly dimensioned.

Widespread belief is that projects concerned with C^3 systems fail for the reason that it is very difficult, if not unfeasible, to develop fault-free software whenever the complexity involved exceeds some «reasonable» threshold.

This is, we believe, a biaised and erroneous view of reality. A major cause of such failures is the lack of a rigorous methodology for designing and dimensioning C^3 systems. If we look back at history, empirical Systems Engineering approaches have been deemed «acceptable» for the reason that, until the mid-80's, application requirements, in terms of criticality and complexity, were not as stringent as they now are. Furthermore, quite often, system failures could be kept hidden from the public or from users, thanks to the «man in the (command-and-control) loop», which is rapidly becoming an elusive concept. Empirical Systems Engineering approaches are a thing of the past. In the case of C^3 systems, it is important to acknowledge the need to address three Computer Science areas altogether, namely Real Time, Fault Tolerant and Distributed Computing. Hence the name of the methodology introduced in section 3, TRDF, which stands for Technologies for Real-time, Distributed, Fault-tolerant computing.[1]

2.2. C^3 systems

Criticality is defined in reference to «catastrophes» resulting from system failures (e.g., loss of property, injury, death, financial loss, environmental damage). Critical systems are typically characterized by accepted probabilities of catastrophic failures which are infinitesimally small (e.g., 10^{-9}/flight-hour for a civilian aircraft). *Complexity* is partially determined by application semantics. Non technical requirements may influence complexity, as would be the case with, e.g., requirements of modularity, reusability and portability of application software components -- referred to as application components thereafter.

More to the point, there is no doubt that having to fulfil proof obligations while solving a C^3 system design/ dimensioning problem leads to increased complexity, compared to that faced when following empirical approaches. Conversely, without any doubt either, the price to be paid at design time so as to cope with increased complexity is way smaller than tolls exacted by catastrophes and/or project cancellations. Remember that provably correct designs are generic, i.e. the manpower invested in design stages is paid once only, which means that a generic solution $<D, d>$ is available «for free» whenever the matching generic problem $<\Lambda, \lambda>$ is to be solved again.

It is well known that formal methods are to be used in the case of C^3 systems. However, it is also recognized that there currently are severe limitations in applying existing formal methods when dealing with *concurrency*, or with *asynchrony*, or with *real-time* constraints or with *fault-tolerance* [1]. Unfortunately, with C^3 systems, these issues arise *altogether*.

Divide-and-conquer approaches are known to be appropriate to master complexity. A C^3 system should thus be viewed as built out of components, each exhibiting a complexity level that is tractable with existing trustable S/W and H/W Engineering methods. This partitioning principle fits well with increased demand for modularity and reusability, as well as with formal approaches based on compositionality (e.g., [2]). Furthermore, there should be a strict separation between application components and system components. System and application components encapsulate algorithms, which are essential w.r.t. the proof obligations. In particular, the role of system components is to endow any collection of application components with those desired individual and global properties required to prove that $<\Lambda, \lambda>$ is satisfied. Design decisions for system components should be kept "orthogonal" to those which are strictly application dependent.

Computer Science has generated a great many algorithms that, for given models, endow individual components as well as collections of components with all those properties that should be exhibited by C^3 systems (see section 3.1). It seems therefore reasonable and useful to take advantage of the proofs that have been established for such algorithms and models.

Most C^3 system design problems being NP-hard, **it is necessary to explictly include specific algorithms in designs**. Algorithms have the virtue of "breaking the complexity", a well established example being that of on-line concurrency control algorithms. With such algorithms, every possible run of any set of concurrently executing application components is proved to be serializable, without having to resort to an exhaustive exploration of the system state space, which is in general a problem of exponential complexity.

3. The TRDF methodology

3.1. Logical and physical properties

In a specification $<\Lambda, \lambda>$, Λ includes, in particular, a set stating which are the *logical* properties sought. In the case of C^3 systems, logical properties sought are combinations of *safety* (a system never enters «bad» states), *liveness* (progress is guaranteed) or *termination*, *timeliness* (activation and termination of application components within specified time windows) and *dependability* (correctness in the presence of

1. TRDF also is the French name of the methodology, which stands for «Temps Réel, Traitement Distribué, Tolérance aux Fautes» (T as a common factor)

partial failures).

Let us illustrate timeliness properties. They can be defined as the Cartesian product of two sets, a class set and a type set. Examples of classes are latest deadlines, bounded jitters. Examples of types are constants, linear functions (of system/environment parameters), non-linear functions. An example of the product "latest deadlines x non-linear functions" for avionics would be deadline = β (altitude)^2, β a constant.

For the sake of simplicity, let us equate Λ with logical properties.

Similarly, λ includes a set stating which are the models under which properties Λ should hold. Models involved are, e.g., models of application components/tasks (sequence, star, tree, directed graph), models of concurrent computations (synchronous, partially synchronous, asynchronous), models of event arrival laws (periodic, sporadic, aperiodic, arbitrary -- the latter defined via bounded densities -- see fig. 1), models of failures (crash, omission, timing, in the time domain; correct/incorrect computations in the value domain). For the sake of simplicity, let us equate λ with such models.

Physical properties simply are valued logical properties (e.g., response times, reliability, availability, throughput). They can be derived from a valuation of $<\Lambda, \lambda>$.

Partial or total orders can be defined over sets of properties/models. Let symbol $\sqsupseteq$ stand for the "dominance" relation. A $\sqsupseteq$ B means that A is equal to or more general than B. Examples of total orders are as follows:

directed graphs $\sqsupseteq$ trees $\sqsupseteq$ stars $\sqsupseteq$ sequences,
bounded jitters $\sqsupseteq$ latest deadlines,
non-linear functions $\sqsupseteq$ linear functions $\sqsupseteq$ constants,
asynchronous $\sqsupseteq$ partially synchronous $\sqsupseteq$ synchronous,
arbitrary $\sqsupseteq$ sporadic $\sqsupseteq$ periodic,
timing $\sqsupseteq$ omission $\sqsupseteq$ crash.

3.2. When is a solution applicable to a problem

Imagine that the Research/R&D community has (partially) explored the space that contains all generic Systems Engineering problems. Whenever a system provider is considering a generic problem $<\Lambda, \lambda>$ derived from a client originated ITT, the question arises as whether, among all generic solutions that have been established previously, one of them at least applies to $<\Lambda, \lambda>$.

A solution that solves a generic problem $<\Lambda', \lambda'>$ is also a solution for $<\Lambda, \lambda>$ whenever the following two conditions are satisfied: $\Lambda' \sqsupseteq \Lambda$ and $\lambda' \sqsupseteq \lambda$.

The first condition simply says that one cannot pick up a solution that endows a system with properties weaker than those specified. The second condition is less well understood. It says that a solution that yields properties $\Lambda' \sqsupseteq \Lambda$ can be considered if and only if these properties have been established under assumptions λ' that are not more restrictive than those specified.

It is easy to check that this condition is frequently violated, especially in scientific papers that address "hard" real-time issues. There are numerous papers containing claims that scheduling method X is so "general" that it can solve almost every problem arising with distributed real-time computations. What is kept hidden, or not acknowledged as being as severe restriction, is the fact that corresponding timeliness properties hold true only for a particular event arrival model, such as periodic arrivals, an obvious violation of the second condition given above whenever specified λ refers to any of the other event arrival models.

Rate-monotonic ("generalized" or not) is a typical example of such a method unduly "marketed" as being a "general" solution. Rate-monotonic does not apply in the case of, e.g., arbitrary arrivals, an arrival model that is much more realistic than that of periodic arrivals. It is easy to understand why clients pick up an arbitrary arrival model, when offered a choice. They are being asked to predict the future, in that they are in charge of providing ω, a description of environment/"adversary" E. Why would they "kill themselves" by pretending that, say in year 2000, E will trigger events strictly periodically?

3.3. Proofs of properties

We do not elaborate on **proofs of safety or liveness**, as they are reasonably well known. Such properties can be established only if **some on-line decision making algorithm** is considered. For example, serializability (a safety property) cannot be enforced in distributed systems without resorting to a concurrency control algorithm or any equivalent algorithm.

A **timeliness proof** has two parts. One consists in expressing **computable functions B that give upper bounds on response times**, for every possible activation of every application component, under given component/ computational/arrival models. This can be done only if **a scheduling algorithm is considered**. The other part consists in expressing those (sufficient, necessary and sufficient) feasibility conditions under which bounds B are valid.

Examples of timeliness proofs are (i) the optimality proof of Earliest Deadline First for a non-overloaded preemptable processor [3], (ii) the optimality proof of D-Over for an overloaded preemptable processor [4].

A **dependability proof** has two parts as well. One consists in demonstrating the existence of such properties as safety (at least), or safety and liveness, under given failure models and failure densities. This can be done only if **some on-line decision making algorithm** is considered, such as, e.g., a reliable broadcast or a group membership algorithm. The other part is concerned with expressing feasibility conditions or **computable functions R that give lower bounds on redundancy degrees** under which safety/liveness properties hold true.

Examples of dependability proofs are the demonstrated correctness of some consensus algorithms in any of the three computational models. An example of function R is R = 3t+1, the smallest number of processors required to tolerate up to t Byzantine processors in synchronous models. An example of feasibility conditions are the weakest failure detector properties in asynchronous models [5].

Functions such as B or R are referred to as *behavioral functions*. Of course, in addition to B and R, other behavioral functions may be needed.

3.4. Provably correct generic designs

Let us assume that generic problem $\langle \Lambda, \lambda \rangle$ has not been solved yet. A simplified view of the stages followed to produce a provably correct generic solution <D, d> is given figure 3.

A design decision, denoted Δ, essentially consists in choosing an assumption set, denoted γ, and a composite algorithm, denoted A*. A composite algorithm endows a system with some combination of the logical properties presented in section 3.1. Let $\Gamma(A^*)$ be those (proved) properties enforced by A*, under γ.

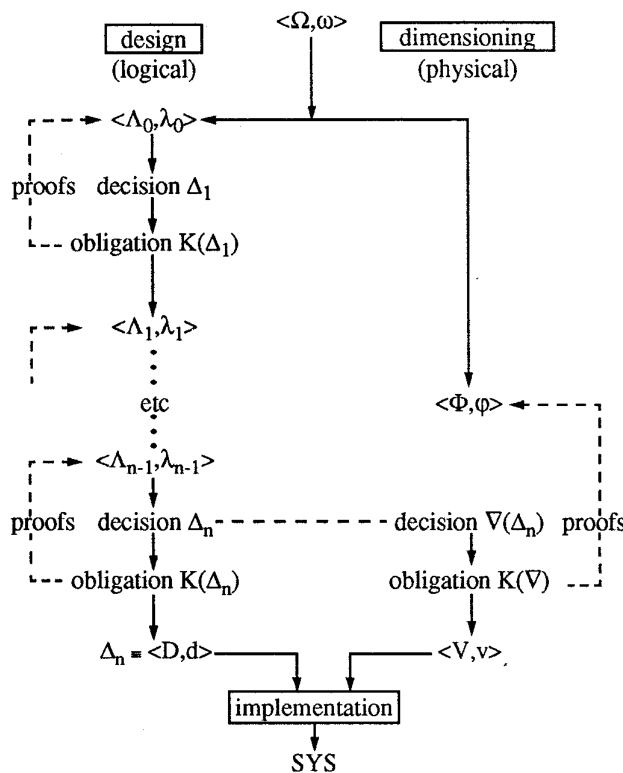

Figure 3. Systems Engineering and the TRDF methodology

<u>Δ-correctness proof obligation $K(\Delta)$</u>

A *design* $\Delta(A^*)$ is *provably correct* if and only if the following conditions are satisfied:

$$\Gamma(A^*) \supseteq \Lambda \text{ and } \gamma \supseteq \lambda .$$

$\langle \Lambda, \lambda \rangle$, the initial problem specification, i.e. the agreed upon capture of $\langle \Omega(l), \omega(l) \rangle$, is denoted $\langle \Lambda_0, \lambda_0 \rangle$. A design decision Δ_1 is made. It is impossible to make and examine design decision Δ_2 as long as proof obligation $K(\Delta_1)$ has not been fulfilled satisfactorily. This is so for the simple reason that the specification of problem $\langle \Lambda_1, \lambda_1 \rangle$ is a result of meeting proof obligation $K(\Delta_1)$. Design decisions and design correctness proof obligations are applied a number of stages, until a level amenable to implementation is reached. This level can be determined by any kind of constraint/ consideration (e.g., imposed or favored H/W, convenient off-the-shelf technology).

As every design in the chain satisfies the Δ-correctness proof obligation, <D, d> provably satisfies $\langle \Lambda, \lambda \rangle$.

3.5. Provably correct dimensioning

A dimensioning, denoted by the operator ∇, is provably correct if it can be demonstrated that <V, v>, a valuation/ dimensioning of <D, d>, satisfies $\langle \Phi, \varphi \rangle$, the valuation of $\langle \Lambda, \lambda \rangle$ according to a pair $\langle \Omega(p), \omega(p) \rangle$ provided by a client.

A dimensioning of <D, d> *is not an implementation* of <D, d>.**A dimensioning ∇ consists in assigning values to implementation variables appearing in behavioral functions.** Examples of such variables are processor speeds, lower/upper bounds on message passing delays, memory capacity, degrees of redundancy in processor groups, etc. The discovery of a correct ∇ usually is an iterative process, even when a priori decisions are made (e.g., prescribed use of a specific type of processor).

<u>∇-correctness proof obligation $K(\nabla)$</u> :

A *dimensioning* ∇ of a design Δ (A*) is *provably correct* *if* $\nabla(\Gamma(A^*)) \supseteq \Phi$ and $\nabla(\gamma) \supseteq \varphi$.

At iteration 1, if both conditions are satisfied, a designer should consider a less «costly» (i.e., less powerful) dimensioning of <D, d>. Such «backward» iterations are repeated until one of the $K(\nabla)$ conditions at least is not satisfied. When this happens, say with ∇_{i+1}, one can conclude that ∇_i is the least «costly» provably correct dimensioning of <D, d>. (This does not imply that ∇_i is the optimal dimensioning, unless <D, d> is a provably optimal design (i.e., every Δ comprised in <D, d> is optimal)).

In the case one condition at least is not satisfied initially, a designer should consider a more «costly» dimensioning of <D, d>. Such «forward» iterations are repeated until both

conditions are satisfied, yielding also the least «costly» provably correct dimensioning of <D ,d>. A dimensioning translates into real costs. If a provably necessary and sufficient dimensioning ∇ is found to be too costly, this is an indication that (i) either the generic problem considered, valued as per <Φ, φ>, has no provably correct solution, given the technology that is affordable or accessible, (ii) or other designs Δ must be considered. The latter outcome should occur less and less often as our accumulated knowledge about provably optimal designs increases with time.

It is only after a provably correct dimensioning has been identified that <V, v> can be established. The pair <<D,d>, <V, v>> is the specification of a system SYS that is proved to behave correctly provided that environment E behaves as specified, logically as per λ, physically as per φ.

Too often, dimensioning is addressed in ad-hoc ways. This is the case whenever arbitrarily simple computational models (e.g., zero-delay or constant-delay abstractions, sequential fault-free computing) are relied upon for the sake of facilitating correctness proofs. The essential issues of asynchrony, concurrency, real-time and fault-tolerance are mostly ignored, left to be addressed by those in charge of implementing some utterly simple models.

This leaves implementors with the job of designing, dimensioning and implementing the equivalent of a distributed, real-time, fault-tolerant «executive». Very often, the implementors' sole obsession is to devise an «efficient» (i.e., «fast», «slim») executive, so that SYS would mimic an imposed idealistic computational model (e.g., infinite computational power is available). Such ad-hoc customized executives have two obvious drawbacks, one being that their cost cannot be amortized over many releases, the other one being that they break the «proof chain». At best, proofs that SYS satisfies <<D,d>, <V, v>> are established under clairvoyance assumptions or for specific and simple cases, almost inevitably in violation of principles $\pi_1/\pi_2/\pi_3$ given in the next section.

Another ad-hoc approach is over-dimensioning. This approach being fully empirical, proofs of logical/physical properties simply cannot be established, even for unbounded system budgets.

Implementation

This is the well-known chain of stages that consist in selecting and/or adapting commercial off-the-shelf hardware and software technology (e.g., operating systems or executives, middleware, microcode, processors) or in developing some specific hardware or software technology, such that <<D,d>, <V, v>> is satisfied. Most often, existing formal approaches run into difficulties with specifications of physical properties, such as <V, v>. Recognition that the ∇-correctness proof obligation comes before the implementation stages should help in this respect. As indicated before, implementation issues are not covered by the TRDF methodology.

From a general perspective, the enforcement of the K(Δ) and K(∇) proof obligations is of utmost importance for early detection of faulty design/dimensioning decisions. Being aware of these obligations, a client (e.g., a certifier) can easily and justfully reject a proposal or a request for certification. Would such obligations have been enforced, some «easy-to-understand» and well marketed «solutions» that violate impossibility results would not be in use. Again, when failures will (inevitably) occur, the real cause (algorithms presented as being able to deliver deterministic services, despite a proof that they can only deliver such services probabilistically) will be kept hidden behind alleged software faults.

4. Principles and useful proof techniques

4.1. Principles

Let us first recall that with C^3 systems, the accepted probability of catastrophic failure, denoted ε, usually is infinitesimal. Let σ be the smallest coverage factor of the models involved in a chain of design decisions. An obvious principle can be given :

(π_1) Any proof of property must be based on models such that $\sigma \geq 1-\varepsilon$ holds true.

Issues of concurrency and asynchrony -- often called «distribution» -- arise with C^3 systems. It follows that such systems must be designed in accordance with distributed systems design principles. One basic principle was established as an impossibility result [6,7] :

Global system states are not directly observable.

At best, such states can be reconstructed *a posteriori* by resorting to specific algorithms (i.e. at some cost). In distributed systems, communication delays are variable and queuing phenomena inevitably develop. It is thus all the more obvious that *future* system states cannot be predicted. Failures are additional sources of uncertainty w.r.t. future state values and timings. Hence the principle :

(π_2) Future system state/timing histories cannot be predicted with certainty.

Proofs of logical and physical properties cannot be established unless the environment/"adversary" is restricted. However, it is wise to avoid considering artificially restricted "adversaries". For example, it is impossible to predict every possible pattern of arrival times for external events. They follow unpredictable arrival laws (think of threats or physical phenomena). Hence the principle :

(π_3) Advance knowledge of future environmental scenarios is limited.

Such assumptions as constant-delay communications or such artifacts as «period enforcers» (the rate-monotonic approach) violate principle π_2. So called «time-triggered» models or periodic arrivals assumptions are antagonistic with principle π_3.

Clairvoyance postulates contradict both π_2 and π_3. Solutions aimed at C^3 systems and developed under such postulates either are inapplicable (such solutions violate principle π_1) or are incorrect.

4.2. Nature of some proof techniques

For the sake of conciseness, we will only sketch out the principles of some the proof techniques used to satisfy the $K(\Delta)$ and $K(\nabla)$ correctness proof obligations imposed with the TRDF methodology. From principles $\pi_1/\pi_2/\pi_3$, one derives the obvious conclusion that optimal solutions can only rest on *on-line algorithms*. With NP-hard problems, one has to sacrifice optimality whenever algorithms of polynomial complexity only can be considered. In that case, partial off-line computations can be contemplated.

Correct solutions rest on *composite* algorithms, which enforce multiple properties altogether. Such properties can be obtained by exploiting state-of-the-art algorithms in Concurrency Control [8], Scheduling and Fault-Tolerance [9] areas. The obligation of addressing these three areas altogether was identified a few years ago [10].

It must be understood that we are shooting for proofs which characterize *deterministic behavior at some boundary conditions in the presence of partial knowledge (of the future, in particular).* Consequently, we cannot expect solving our problems by resorting to probabilistic approaches (e.g., Queueing theory) or statistical approaches (e.g., simulation) which are nevertheless useful to predict average behaviors. The intent of the approach advocated here is to «let the probabilities in» only when a client is asked to postulate physical environmental scenarios, that is *after design*, rather than «let them in» when a designer establishes (design) correctness proofs. Reluctance to use on-line algorithms with C^3 systems stems from the erroneous belief that future system behaviors would become unpredictable with such algorithms. This is a strange view, given that so many results proving the opposite have been published in the past.

We have used pure on-line algorithms to solve the following distributed scheduling problem [11] : A distributed multiaccess broadcast channel is shared by a number of message sources. The exact number of sources is unknown but bounded. Message arrival laws are arbitrary (only upper bounds on arrival densities are given). Messages are to be transmitted within strict deadlines, revealed upon message arrivals only. Give A*, Γ(A*) and feasibility conditions.

A distributed message scheduling algorithm A* being chosen or devised, a timeliness proof consists in giving the expression of function B(i,r), the upper bound on service times for a message ranked r-th in source i's waiting queue, $\forall$r, $\forall$i. Such bounds are obtained using adversary arguments. In [11], we have considered an arbitrarily devilish adversary which is provided with an infinite amount of messages. The game imposed upon the adversary is a deterministic variant of Ethernet, called Deadline-Oriented-Deterministic CSMA-CD. Clearly, this problem cannot be solved with solutions based on off-line computations, such as [12, 13]. Such solutions are falsely reassuring. Their coverage factor σ is provably inferior to $1-\varepsilon$ for most C^3 systems. In [14], we demonstrate that optimal solutions to the problem introduced in [11] can only be in class Non-Preemptive Earliest-Deadline-First/Collision-Detection-and-Resolution (that is, Token-Passing cannot be optimal).

Using the TRDF methodology, we have been able to deliver a specification $\langle D, d\rangle$ of a generic problem $\langle \Lambda, \lambda\rangle$ derived from an ITT that resembles the generic example shown fig.1. We have also delivered the specification of a corresponding feasibility Oracle.

Proof techniques used for this work have been drawn from different disciplines, such as conventional analytical calculus, conventional scheduling theory, adversary arguments and computations on graphs in a specific algebra.

From a more general perspective, reasoning under uncertainty is known to be possible, as amply demonstrated by e.g., Game theory or Decision theory. Optimality of composite on-line algorithms can be established by resorting to Competitive Analysis [4, 15].

5. Conclusion

The major aims of this paper are, (i) to contribute to a better understanding of the nature of some issues which do not seem to be rigorously addressed with the design and the dimensioning of C^3 systems, a prerequisite to identifying the correct solutions, (ii) to introduce the TRDF methodology which seems to be an early Systems Engineering methodology for C^3 systems that is based on mathematical reasoning. The prominent part played by the provably correct generic design and provably correct dimensioning stages in the lifecycle of a (possibly to-be-certified) C^3 system have been emphasized.

Combining formal methods (for specifying initial application requirements, generic design problems, generic solutions) with the TRDF methodology is believed to be a very promising approach to the development of correct and trustable C^3 systems. There is a wide open field of research and experiments to be conducted by our community so as to provide clients with provably correct C^3 systems.

References

1. J. Rushby, Formal Methods and the Certification of Critical Systems, SRI-CSL Technical Report 93-07 (Nov. 1993) 308 p.
2. J.R. Abrial, The B Method, Prentice Hall pub., (1994).
3. C.L. Liu, J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, J.ACM, 20-1 (Jan. 1973) 46-61.
4. S. Baruah et al., On-Line Scheduling in the Presence of Overload, 32nd Symp. on Foundations of Computer Science (1991) 100-110.
5. T.D. Chandra, V. Hadzilacos, S. Toueg, The Weakest Failure Detector for Solving Consensus, ACM PODC-11 (Aug. 1992) 147-158.
6. G. Le Lann, Distributed Systems - Towards a Formal Approach, IFIP Congress, North-Holland pub. (1977) 155-160.
7. L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, Com. ACM, 21-7 (July 1978) 558-565.
8. C.H. Papadimitriou, The Theory of Concurrency Control, Computer Science Press, (1986) Rockeville (MD).
9. J.C. Laprie (ed.), Dependability : Basic Concepts and Terminology, vol. 5 of Dependable Computing and Fault-Tolerant Systems, Springer-Verlag (1991).
10. G. Le Lann, Distributed Real-Time Processing, in Computer Systems for Process Control, R. Güth ed., Plenum Press pub. (1986) 69-88.
11. G. Le Lann, N. Rivierre, Real-Time Communications over Broadcast Networks : the CSMA-DCR and the DOD/CSMA-CD Protocols, INRIA Res. Report 1863 (1993) 35 p.
12. H. Kopetz, G. Grünsteild, TTP - A Protocol for Fault-Tolerant Real-Time Systems, IEEE Computer (Jan. 1994) 14-23.
13. L. Sha, S.S. Sathaye, A Systematic Approach to Designing Distributed Real-Time Systems, IEEE Computer (Sept. 1993), 68-78.
14. J.F. Hermant, G. Le Lann, N. Rivierre, «A General Approach to Real-Time Message Scheduling over Distributed Broadcast Channels», IEEE/INRIA Conf. on Engineering Technologies and Factory Automation, (1995), 191-204.
15. R.M. Karp, On-Line Algorithms Versus Off-Line Algorithms : How Much is it Worth to Know the Future ?, IFIP Congress, Vol I, Elsevier pub. (1992) 416-429.