Predictability in Critical Systems

Gérard Le Lann

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

Abstract. Predictability is crucial in critical applications and systems. Therefore, we examine sources of uncertainty for each of the four phases that span a project lifecycle, from initial problem capture, to system implementation, when conducted according to proof-based system engineering principles. We explore the concept of coverage applied to problems, solutions, assumptions, along with a generic problem that arises with critical applications such as, e.g., air traffic control/management, namely the real-time uniform atomic broadcast problem. We examine two design styles, namely asynchronous and synchronous solutions, and compare the resulting assumptions as well as their coverages. The central issues of overloads and timing failures that arise with synchronous models are investigated in detail.

1 Introduction

We consider X-critical applications and systems, where X can be any such qualifier as, e.g., life, mission, environment, business or asset. Intuitively, an application or a system is critical whenever violating a specification may lead to "catastrophes". Therefore, levels of "confidence" set for such applications or systems are extremely high. For instance, in the case of air traffic control/air traffic management (ATC/ATM), accumulated time durations of inaccessibility to critical services should not exceed 3 seconds a year, which translates into an upper bound of 10^{-7} on acceptable unavailability of critical services. With stock trading or with some defense applications, response times of less than 2 seconds (a timeliness property) are required, 10^{-4} being an upper bound on acceptable rates of lateness. Acceptable failure rates of processors for urban trains should be less than 10^{-12} per hour.

Let Ω be some assertion. Coverage(Ω) is defined as the probability that Ω is true. Assumption coverage has been introduced in [17]. Assumptions relate to the environment where a system is to be deployed or to implementation. We extend the notion of coverage to properties, to problems and to solutions. Such coverages are goals set for designers. For instance, with ATC/ATM, coverage (*availability property for critical services*) should be *shown to be* as high as $1 - 10^{-7}$. Note that an assumption coverage also is a goal set for those who are in charge of "implementing the assumption".

A.P. Ravn and H. Rischel (Eds.): FTRTFT'98, LNCS 1486, pp. 315–338, 1998.
© Springer-Verlag Berlin Heidelberg 1998

Endowing critical systems with specified properties that hold with such high coverages is a notoriously difficult task. Most often, critical application problems translate into combined real-time, distributed, fault-tolerant, computing problems. Only a small subset of such problems has been explored so far.

There are many examples of systems that have failed to meet such stringent requirements. There is no general agreement on what are the dominant causes of such failures. In the recent past, widespread belief has been that software - noted S/W - must have become the major cause of failures, given that hardware - noted H/W - is so "reliable". However, a growing number of analyses and studies demonstrate that such is not the case. For instance, a study of the failure reports concerning the US Public Switched Telephone Network (PSTN) establishes that S/W caused less downtime (2%) than any other source of failure except vandalism [8]. Overloads are the dominant cause (44%) of downtime measured as the product *number of customers affected·outage duration*. Overloads that occur "too frequently" result into "bad" coverages of such properties as availability or timeliness, which is barely acceptable for the PSTN, and certainly unacceptable for a business-critical system. It turns out that a vast majority of faults that cause overloads are neither S/W originated nor H/W originated. They are system engineering faults.

We have previously argued that, as is the case with H/W engineering and, more recently, with S/W engineering, proof-based approaches (based upon informal or, better, formal methods) should become the rule in system engineering [10]. In this paper, we will make the (extraordinary) assumption that S/W components and H/W components are faultless, i.e. that they actually implement their specifications. Systems would still fail, for the following reasons: (i) specifications may be flawed, due to faulty design and/or dimensioning decisions, (ii) residual assumptions may be violated at run-time.

Ideally, only the initial application problem specification would possibly turn out to be flawed, given that the future cannot be predicted with certainty. Any subsequent specification should be correctly derived from this one, via system design and dimensioning decisions assorted with proofs. With stochastic system engineering (SE) approaches, coverages are involved with design and dimensioning decisions. In this paper, we consider deterministic SE approaches (deterministic algorithms, deterministic analyses). Then, only the coverage of the initial specification does matter.

With critical applications, deterministic proof-based SE is appropriate for an additional reason: Proofs of correctness must be established for worst-case operational conditions, worst-cases being embodied, albeit not explicitly, in an application problem specification.

Residual assumptions (RA) are those needed to prove implementation correctness. Arbitrarily fast faultless sequential processors, or byzantine processors, or absence of H/W metastable states, or bounded broadcast channel slot times, are examples of common RAs. Their coverages can be computed or estimated, by resorting to, e.g., analytical modelling or statistical analysis. Only RAs that have "very high" coverages are acceptable for critical systems - clearly not the

case with the first example. Consequently, provably correct system design and dimensioning phases must be conducted until the RAs are acceptable.

Under our assumption, and proof-based SE principles being followed, the only coverages - i.e. uncertainties - left relate to, (i) the capture of an application problem and, (ii) the implementation of residual assumptions.

The basic principles of proof-based SE are introduced in Section 2. In Section 3, we examine sources of uncertainty for each of four important phases that span a project life-cycle, from initial problem capture, to system implementation. Elegant solutions, or efficient solutions, may rest on residual assumptions that have low coverages. For the sake of illustration, in Section 4, we consider a generic problem that arises with critical applications such as, e.g., ATC/ATM, namely the real-time uniform atomic broadcast problem, examine two design styles (asynchronous and synchronous solutions), and compare resulting residual assumptions.

This leads us to revisit popular justifications of asynchronous models, as well as popular approaches to the implementation of synchronous assumptions, to find out that some have low coverages. The central issue of timing failures is explored in Section 5. It is shown that dominant causes of timing failures, namely overloads and excessive failure densities, can be dealt with more carefully than is usually asserted.

2 Proof-Based System Engineering

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, to produce a global and implementable specification of a system (noted 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.

The notation $\langle Y \rangle$ (resp. $[Z]$) is used in the sequel to refer to a specification of a problem Y (resp. a solution Z). Notation $\langle y \rangle$ (resp. $[z]$) is used to refer to a specification of a set of variables which parameterize problem Y (resp. solution Z). 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 life-cycle, namely the problem capture phase, the system design phase, and the system dimensioning phase (see figures 1 and 2, where symbol $\Rightarrow$ stands for "results in"). Proof-based SE also addresses phases concerned with changes that may impact a system after it has been fielded, e.g., modifications of the initial problem, availability of new off-the-shelf products. Such phases simply consist in repeating some of the three phases introduced above, and which pre-

cede phases covered by other engineering disciplines (e.g., S/W engineering), which serve to implement system engineering decisions.

2.1 The Problem Capture Phase

This phase is concerned with, (i) the translation of an application problem description into $\langle A \rangle$, which specifies the generic application problem under consideration and, (ii) the translation of $\langle A \rangle$ into $\langle X \rangle$, a specification of the generic computer science problem that matches $\langle A \rangle$. A generic problem is an invariant for the entire duration of a project.

Specifications $\langle A \rangle$ and $\langle X \rangle$ are jointly produced by a client and a designer, the latter being in charge of identifying which are the models and properties commonly used in computer science whose semantics match those of the application problem. Consequently, a specification $\langle X \rangle$ actually is a pair $\{\langle m.X \rangle, \langle p.X \rangle\}$, where m stands for models and p stands for properties.

For example, statement "*workstations used by air traffic controllers should either work correctly or stop functioning*" in $\langle p.A \rangle$ would translate as "*workstations dependability property is observability = stop failure*" in $\langle p.X \rangle$. See Section 4 for examples of models and properties.

Variables appear in specifications $\langle A \rangle$ and $\langle X \rangle$. Let us focus on $\langle X \rangle$. Notation $\langle x \rangle$ is used to refer to a specification of those variables in $\langle X \rangle$ that are left unvalued. As for $\langle X \rangle$, $\langle x \rangle$ is a pair $\{\langle m.x \rangle, \langle p.x \rangle\}$.

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

Fig. 1. Problem Capture

The genericity degree of $\langle X \rangle$ may vary from 0 ($\langle x \rangle$ is empty) to ∞ (every variable in $\langle X \rangle$ appears in $\langle x \rangle$). Of course, any degree of genericity has an associated cost and a payoff.

2.2 The System Design Phase

This phase is to be conducted by a designer. A design phase has a pair $\{\langle X \rangle, \langle x \rangle\}$ as an input. It covers all the design stages needed to arrive at $[S]$, a modular specification of a generic solution (a generic 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.

More precisely, one solves a problem $\{\langle m.X(.)\rangle, \langle p.X(.)\rangle\}$ 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.

Subproblems result from a modular decomposition. Fulfilling a design correctness proof obligation (see Section 2.6) guarantees that if every subproblem is correctly solved, then the initial problem is correctly solved as well by "concatenating" the individual solutions, which eliminates those combinatorial problems that arise whenever such proof obligations are ignored. And so on. Consequently, a design phase has its stages organized as a tree structure. By the virtue of the uninterrupted tree of proofs (that every design decision is correct), $[S]$ - the union of those specifications that sit at the leaves of a design tree - provably correctly satisfies $\langle X\rangle$. If $\langle X\rangle$ is a correct translation of $\langle A\rangle$, then, by transitivity, $\langle A\rangle$ 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 some other designer.

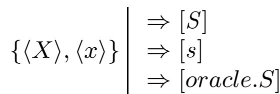

Fig. 2. System Design

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. Feasibility conditions (FCs) are analytical expressions derived from correctness proofs. For a given architectural and algorithmic solution, they define a set of scenarios that, with certainty, includes all worst-case scenarios that can be deployed by "adversary" $\langle m.X(.)\rangle$. FCs link together these worst-case scenarios with computable functions that serve to model properties stated in $\langle p.X(.)\rangle$. 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 $\{\langle X\rangle, [S]\}$-dependent component of a more general system dimensioning tool.

Lack of FCs is an important source of failures for critical systems, as demonstrated by, e.g., the system shutdowns experienced with the Mars PathFinder probe.

2.3 The System Dimensioning Phase

The purpose of a dimensioning phase is to find a valuation $V([s])$, i.e. a quantification of system S unvalued variables, such as, e.g., sizes of memory buffers, sizes of waiting queues, processors speeds, databuses throughputs, number of databuses, processors redundancy degrees, total number of processors.

$V([s])$ must satisfy a particular valuation $V(\langle x \rangle)$, i.e. a particular quantification of the captured problem-centric models and properties, which is - directly or indirectly - provided by a client.

$V(\langle x \rangle)$: input to *oracle.S* $V([s])$: output from *oracle.S*

Fig. 3. System Dimensioning

One or several dimensioning phases may have to be run until $[oracle.S]$ declares that there is a quantified S that solves a proposed quantified problem $\{\langle X \rangle, V(\langle x \rangle)\}$ (or declares that the quantified problem considered is not feasible). How many phases need to be run directly depends on the genericity of $[S]$. Consider for example that $[s]$ is close to empty, which happens whenever it is decided a priori that S must be based on specific off-the-shelf or proprietary products. The good news are that a small number of dimensioning phases need to 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.

The pair $\{[S], V([s])\}$ is a modular specification of a system S that provably solves problem $\{\langle X \rangle,\ V(\langle x \rangle)\}$. Modules of $\{[S], V([s])\}$ are contracts between a (prime) designer and those (co/sub) designers in charge of implementing S.

2.4 The System Implementation Phase

If deemed implementable, the set $\{[S], V([s])\}$ is the borderline, i.e. the interface, 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, which result from system engineering work. It follows that it is 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 lead to incorrect systems.

Too often, system failures due to SE faults are mistakenly believed to originate in S/W design or S/W implementation errors. See [8], [12] and [13] for counter-examples.

2.5 Final Comments

Models, properties, and algorithms, can be organized into classes. Furthermore, it is possible to structure every class after a hierarchy or a partial order. See [17] for an example with the class of failure models. We will use the terminology introduced in [10]. An element that precedes another one in a hierarchy or a partial order will be said to be weaker than its successor (successor is stronger). As for models, this is so for the reason that the set of predecessor runs (e.g., behaviors, traces) is included in the set of successor runs. For instance, the *byzantine failure* model [9] is stronger (albeit more "permissive") than the *omission failure* model (which is more "restrictive"). Byzantine failure behaviors include omission-only failure behaviors. The converse is not true.

This is so for properties for the reason that they result from explicitly restricting runs to those of interest (unlike models). For instance, *causal atomic broadcast* is stronger than *FIFO broadcast* [6]. Similarly, *bounded jitters* is stronger than *latest termination deadlines* [10].

Design correctness proof obligations result from class structuring. A solution that solves a problem $\{\langle m.X\rangle, \langle p.X\rangle\}$ is a correct design solution for a problem $\{\langle m.X'\rangle, \langle p.X'\rangle\}$ if $\langle m.X\rangle$ (resp. $\langle p.X\rangle$) is stronger than $\langle m.X'\rangle$ (resp. $\langle p.X'\rangle$). Given some appropriate metrics, one can similarly define dimensioning correctness proof obligations. See [10] for more details.

Note that there are differences between the various sets of assumptions involved with a pair $\{\langle X\rangle, [S]\}$.

The specification $\langle m.X\rangle$ (resp. $V(\langle m.x\rangle)$) states *problem* (resp. *valuation of problem*) assumptions. In many cases, these specifications are predictions on future operational conditions, which must have some coverages.

Specification $[m.S]$ (resp. $V([m.s])$) states *design* (resp. *valuation of design*) residual assumptions (design tree *leaves*), which are to be "implemented". Note that they are the only design (and valuation of design) assumptions whose coverages must be estimated. Indeed, design assumptions embodied in a design tree (design tree *nodes*) are necessarily correctly "implemented", as well as their valuations, a result of fulfulling design and dimensioning correctness proof obligations.

Let $C(X, V(x))$ be the coverage of quantified problem $\{\langle X\rangle, V(\langle x\rangle)\}$. In other words, $Pr\{\langle m.X\rangle$ or $V(\langle m.x\rangle)$ is violated$\} \leq 1 - C(X, V(x))$. This is the coverage of predictions - i.e. assumptions - relative to some future operational environment.

To assess the intrinsic quality of a solution, shown to enforce properties $\langle p.X\rangle$ valued as per $V(\langle p.x\rangle)$, one needs to eliminate from our scope of consideration those uncertainties due to the impossibility of telling the future. The validity of predictions regarding models or their valuations is not discussed. Coverage $C(S, V(s))$, which is the coverage of quantified solution $\{[S], V([s])\}$ for quantified problem $\{\langle X\rangle, V(\langle x\rangle)\}$, serves this purpose.

In other words, conditional probability $Pr\{\langle p.X\rangle$ or $V(\langle p.x\rangle)$ is violated $|$ neither $\langle m.X\rangle$ nor $V(\langle m.x\rangle)$ is violated$\} \leq 1 - C(S, V(s))$.

3 Sources of Uncertainty

3.1 Capture Phase

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" (e.g., 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 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 coverages is void with such models. 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". Picking up the appropriate models for $\langle m.A \rangle$ and $\langle m.X \rangle$ boils down to making tradeoffs between having coverage $C(X, V(x))$ sufficiently close to 1 and retaining models as weak as acceptable.

As for early valuations of problem-centric models (via $V(\langle m.x \rangle)$), comments made for the dimensioning phase apply (see further).

3.2 Design Phase

Coverages do not apply to deterministic design approaches. Properties such as, e.g., serializability, timeliness, are either not ensured or they are, under specific feasibility conditions. There are no "proof coverages" for proofs of properties or feasibility conditions (e.g., lower bounds on a redundancy degree or on a number of modules, upper bounds on response times), even if solutions considered are "complex", possibly due to the fact that strong models appear in $\langle m.X \rangle$.

Fulfilling design correctness proof obligations guarantees that properties and models considered during a design phase are at least as strong as those stated in $\langle X \rangle$. Hence, there are no coverages involved with whether $[S]$ solves $\langle X \rangle$.

This is not necessarily the case with popular design styles. For example, with the problem described in Section 4, one ends up with having to "implement" a synchronous communication module, referred to as Net. Picking up the periodic arrival model for messages submitted to Net would be invalid, given that some of the tasks which generate these messages are activated according to a unimodal arbitrary model, which is stronger than the periodic model. It is legal only to consider unimodal or multimodal arbitrary models. Finding optimal or "good" solutions under such models is definitely more intricate than when considering periodic arrivals, an assumption made by many authors, presumably for the reason that these models are more tractable.

Residual design assumptions $[m.S]$ and $V([m.s])$ must be shown to be implementable with "good" coverages. There is an inverse relationship between a "design effort level" and subsequent "implementation effort level".

To continue with the same example: To build Net in such a way that, under unimodal arbitrary arrivals, an upper bound on message delays does hold is not just an "implementation concern". Stated differently: Coverage of residual assumption "synchronous Net exists" is close to 0. This calls for further design work. See [7] for an example where the Hard Real-Time Distributed Multiaccess problem raised with Net is solved considering Ethernets or CableTV networks, technologies whose coverages can be accurately estimated, given their widespread usage, and made very close to 1, given that these technologies are well mastered.

3.3 Dimensioning Phase

Valuations of problem-centric models (via $V(\langle m.x \rangle)$) have related coverages. Assignment of a value to a number of byzantine failures has a coverage, whereas the byzantine model itself has no related coverage.

Under a proof-based SE approach, physical dimensionings of systems have no coverages associated to them. Those numerical values that appear in $V([s])$ specify a "safe" physical dimensioning of a system, a by-product of checking the feasibility of some problem-centric quantification $V(\langle x \rangle)$. Such dimensionings are certainly sufficient (ideally, they should be necessary and sufficient).

Consequently, there are no coverages involved with whether V($[s]$) satisfies V($\langle x \rangle$).

Note that, with critical applications, valuation of $C(S, V(s))$ is usually "frozen" by a client when conducting a capture phase. Hence, most often, $C(S, V(s))$ appears in $\langle p.X \rangle$ only (not in $\langle p.x \rangle$).

3.4 Implementation Phase

Which are the coverages involved with implementing $\{[S], V([s])\}$? Recall our assumption that, thanks to some S/W and H/W formal methods, implementation of the architectural and algorithmic solutions embodied in $\{[S], V([s])\}$ is faultless. Given that, as argued previously, the design and the dimensioning of a solution can be faultless, the only issue left is that of implementing residual assumptions $[m.S]$ and $V([m.s])$.

Let ϕ be the smallest of the coverages involved with residual assumptions. It follows that $C(S, V(s)) = \phi$ under proof-based SE approaches.

It derives from this that it is always safer to consider strong models, which bring ϕ - as well as $C(X, V(x))$ - arbitrarily close to 1. For instance, the byzantine failure model or the asynchronous model cannot be "violated" at run-time, and it is very easy to implement them correctly. However, picking up such models may not be feasible because of, e.g., impossibility results, or may be unjustifiable on financial grounds, given that dimensionings - i.e. system costs - may be linear or superlinear functions of models strength.

Conversely, picking up weak (i.e. restrictive) models - while still satisfying design correctness proof obligations - lowers ϕ or $C(X, V(x))$. This is a source of concern with some conventional approaches to the design and construction of critical systems.

4 A Generic Problem with Critical Applications

For the purpose of illustration, let us examine the real-time uniform atomic broadcast problem, noted $\langle RTUAB \rangle$. This problem arises with applications (e.g., ATC/ATM, stock trading, "risky" process control, transportation, protective defense) that need to be run over distributed and/or redundant real-time systems, for the purpose of guaranteeing continuous and timely delivery of critical services despite partial failures.

For instance, in an ATC/ATM cell, an air traffic controller whose workstation "goes down" has immediate access to another workstation - by physical displacement - which should display exactly what would have been displayed by the failed workstation. Such failures "impacting" a given air traffic controller, i.e. surveillance of a given portion of the sky, may occur a number of times before workstations are repaired or replaced with new ones.

Given that within distributed and/or redundant systems, delays are variable, that concurrency is unavoidable - decisions may be made concurrently by different pilots, by different controllers - and that failures impact portions of a system differently and at different times, it is far from being trivial to ensure that redundant workstations do behave identically at all times, or that common knowledge (e.g., of what the "current sky state" is) is indeed maintained among all workstations, in due time, within a cell.

Non real-time versions of this problem have been considered first, such as $\langle UAB \rangle$, the uniform atomic broadcast problem (see [6]) which, informally, is as follows. In a system of processors, some of which may stop unexpectedly and anonymously, *Messages* are broadcast by processors, possibly concurrently, by invoking an Atomic Broadcast (*Message*) primitive, noted ABroadcast. Correct processors must deliver the same *Messages* in the same order. That is, *Messages* are Atomically Delivered, noted ADelivered.

Let $\mathcal{A}$ be an algorithmic solution. $\langle UAB \rangle$ can be solved in various ways, considering computational models and algorithms ranging from synchronous to asynchronous.

Simply stated, in synchronous models - as defined in [4] and [16] - delays are supposed to be bounded, lower and upper bound values being known, or relative speeds of processors, and of links, are supposed to be bounded, or execution steps are supposed to be taken simultaneously, in lock-step rounds. In section 5, we will also consider that processors have access to synchronized clocks [1].

In asynchronous models - as defined in [5] - any execution step may take an arbitrarily long, but finite, time. In other words, delays cannot be bounded.

It has been shown that $\langle UAB \rangle$ has no (deterministic) algorithmic solution in asynchronous models [1], a result derived from [5]. With asynchronous models that are "augmented" with inaccurate failure detectors - noted *FDs* - that exhibit particular properties [1], $\langle UAB \rangle$ can be solved.

[1] Models where processors have approximate knowledge of time, of bounds, belong to the class of partially synchronous [4] or timing-based [16] models.

Solving $\langle UAB \rangle$ only does not help with critical applications. Informally, what is missing is the requirement that *Messages* that are ABroadcast should be ADelivered "in time". Hence $\langle RTUAB \rangle$.

4.1 Problem $\langle RTUAB \rangle$

Below is a sketch of $\langle RTUAB \rangle$, adapted from a simpler problem investigated within the ATR project [2]. A processing module, referred to as a module, comprises a processor, tasks, data structures, system S/W, on top of which some algorithmic solution $\mathcal{A}$ is run.

$\langle m.RTUAB \rangle$

- Finite set Q of modules, interconnected by a network module referred to as Net. Nominal size of Q is $n > 1$.
- Finite set Θ of tasks. Mapping of Θ onto Q (boolean matrix $\Pi(\Theta, \mathrm{Q})$) is unrestricted.
- A task θ is a finite sequence of code that invokes primitive ABroadcast, once or many times. The longest path of execution (in "code length") is known for every task, noted $x(\theta)$ for task θ.
- Finite set E of external event types. Mapping of E onto Θ (boolean matrix $\Pi(\mathrm{E}, \Theta)$) is unrestricted. Event $e(\theta)$ - an occurrence of some event type - is the arrival of a request for running task θ.
- External event arrivals:
  - sporadic for some event types (sporadicity interval $sp(t)$ for type t),
  - unimodal arbitrary for others (bounded density $a(t)/w(t)$ for type t).
- For some external event types, causal dependencies exist between events.
- Failure model for modules: stop (correct behavior or immediate crash).
- Failure occurrences: system-wide unimodal arbitrary (bounded density f/W for set Q).

$\langle p.RTUAB \rangle$

- Validity: Every *Message* ABroadcast by a correct module is ADelivered by this module.
- Uniform agreement: If *Message* is ADelivered by a module, *Message* is ADelivered by every correct module.
- Uniform integrity: *Message* is ADelivered at most once by every module, and only if *Message* was previously ABroadcast.
- Uniform total order: If $Message_1$ and $Message_2$ are ADelivered by any two modules, they are ADelivered in the same order.

[2] Members of the ATR project are Axlog Ingénierie, Dassault Aviation, École Polytechnique/LIX, INRIA, Thomson-Airsys, Université Paris VII/LIAFA, Université de Grenoble/LMC. The project is supported by Délégation Générale à l'Armement/DSP, Ministère de l'Éducation Nationale, de la Recherche et de la Technologie, and CNRS.

- Uniform external causality: If $Message_j$ causally depends on $Message_i$ and both are ADelivered by a module, $Message_i$ is ADelivered prior to $Message_j$.
- Timeliness: Timeliness constraints are strict relative termination deadlines. A deadline is associated to every task, noted $d(\theta)$ for task θ. If event $e(\theta)$ occurs at time τ, task θ must be completed by time $\tau + d(\theta)$ at the latest.
- Availability: Every task of set Θ should always be runnable within set Q.
- Coverage (solution) = $C(S, V(s))$.

Comments The unimodal arbitrary arrival model is resorted to whenever advance knowledge regarding arrivals is restricted to be an upper bound on arrivals density, noted $a(t)/w(t)$ for event type t, where $w(t)$ is the size of a (sliding) time window and $a(t)$ is the highest number of arrivals of type t events that can occur within $w(t)$.

$Messages$ are arguments of the ABroadcast and ADeliver primitives. These primitives, as well as algorithms $\mathcal{A}$, make use of lower level primitives usually called send and receive, whose arguments are physical messages - referred to as messages in the sequel - processed by module Net. This module is supposed to be reliable, i.e. messages are neither corrupted nor lost.

Variable W is a time window that represents a worst-time-to-repair. For instance, W is an upper bound on a mission duration, an upper bound on a period of regular maintenance/repair activities. The f/W bound is the well known "up-to-f-failures-out-of-n-modules" assumption.

$Messages$ are ADelivered by $\mathcal{A}$ on every module, in a waiting queue, to be read in FIFO order by tasks other than tasks in Θ.

Timeliness constraint $d(\theta)$ specified for task θ induces timeliness constraints for completing each of the ABroadcast invocations (triggered by tasks) and each of the ADeliver operations (performed by $\mathcal{A}$). It is a designer's job to derive a $Message$ timeliness constraint from the invoking task's timeliness constraint. If τ is the arrival time of $e(\theta)$, ADelivery ($Message$) matching the latest ABroadcast ($Message$) triggered by θ must be completed by some time smaller than $\tau + d(\theta)$.

Note that, on every module, the execution of tasks and the execution of $\mathcal{A}$ are sequentially interleaved, i.e. the execution of a task may be suspended (its module is preempted) so as to run $\mathcal{A}$.

Observe that, contrary to common practice, availability is not stated as a probability. Similarly, W is not the usual mean-time-to-repair variable handled under stochastic approaches. Indeed, there is no reason to consider that dependability properties should be of "probabilistic" nature, while other properties (e.g., timeliness or total order) are not. Critical services should always be accessible, delivered on time, and should always execute as specified. Why make a special case with dependability properties?

There are reasons why, ultimately, properties - any property - and/or their valuations may not hold with certainty. A coverage is then specified for each of them, separately. This is not the case considered here.

Coverage $C(S, V(s))$ applies to any of the properties or any of their valuations stated as per $\{\langle RTUAB \rangle, V(\langle rtuab \rangle)\}$.

One possibility for specification $\langle m.rtuab \rangle$ would be that it contains all the variables that appear in $\langle m.RTUAB \rangle$. Ditto for $\langle p.rtuab \rangle$, w.r.t. $\langle p.RTUAB \rangle$, to the exception of $C(S, V(s))$ - see Section 3.3. Another possibility is that matrix $\Pi(\mathrm{E}, \Theta)$, as well as some of the elements of matrix $\Pi(\Theta, \mathrm{Q})$, would not appear in $\langle m.rtuab \rangle$, neither would some of the variables $d(\theta)$ appear in $\langle p.rtuab \rangle$, for the reason that values taken by these variables are known and cannot be changed, i.e. they conceal "frozen" advance knowledge.

4.2 Solutions

Recall that valuations of $\{\langle m.rtuab \rangle, \langle p.rtuab \rangle\}$, of $C(S, V(s))$, are decided upon by a client or an end user.

Proving that $\{\langle RTUAB \rangle, V(\langle rtuab \rangle)\}$ is solved with $\{[S], V([s])\}$ is conditioned not only on proving that the solution specified is correct, but also on showing that ϕ is at least equal to $C(S, V(s))$.

Let us now consider two design solutions, namely a synchronous solution and an asynchronous solution.

Contrary to commonly held beliefs, a critical problem can be solved with a solution based on an asynchronous algorithm designed considering an asynchronous computational model. For example, for solving $\langle RTUAB \rangle$, one can select an asynchronous algorithm that solves $\langle UAB \rangle$ and that has been shown to have the best worst-case termination time (e.g., in number of phases). This algorithm would be transformed so as to internally schedule $Messages$ according to their timeliness constraints, as well as have the total $Message$ orderings of ADeliveries driven by $Message$ timeliness constraints [3]. Doing this would lead to the best (possibly optimal) timeliness-centric FCs for $\langle RTUAB \rangle$.

The important distinction made here is between *computational* models resorted to during a design phase and *system* models considered for conducting dimensioning and implementation phases. This distinction, which derives directly from proof-based SE principles, has been suggested in [11]. For example, an asynchronous computational model is specified in $[m.S]$, for every processing module, as well as for module Net. These modules are "immersed" in synchronous system models. System models must be synchronous, given that one must know, prior to usage, whether FCs are satisfied.

Is it worthwhile to proceed as described despite the fact that, ultimately, a synchronous system is to be deployed? Why not consider synchronous models up front (at design time)? One major advantage of asynchronous solutions is that they are universally portable. No matter which system they run onto, they retain their safety properties. This is not necessarily the case with synchronous solutions. For example, there are synchronous algorithms that would lose some of the safety properties stated in $\langle RTUAB \rangle$ whenever bounds on delays are violated.

[3] Even though neither global time nor absolute deadlines are accessible in an asynchronous system, they can be modelled via (unvalued) variables, which can be used by algorithms to enforce particular schedules.

Conversely, one must keep in mind that some of the properties achieved with asynchronous solutions immersed in a synchronous system, or their valuations, are weaker than those achieved with synchronous solutions. This is the case, in particular, with timeliness properties, shown by the fact that FCs established for an asynchronous solution may be (significantly) more pessimistic than FCs established for a synchronous solution.

Asynchronous Solutions As for $\langle UAB \rangle$, they inevitably rest on such constructs as, e.g., inaccurate *FDs*, which are characterized by their completeness and accuracy properties [1].

It has been shown that $\diamond W$ (weak completeness and eventual weak accuracy) is the weakest class of *FDs* for solving such problems as Consensus or Atomic Broadcast if less than $\lceil n/2 \rceil$ modules fail [2]. Many algorithmic solutions are based on *FDs* - e.g., $\diamond S$ - that can be built out of $\diamond W$. Whenever *FDs* properties happen to hold true in a system, liveness properties enforced by some $\mathcal{A}$ hold true as well.

However, with existing asynchronous algorithms, orderings of computational steps are not driven by timeliness constraints. Furthermore, with *FDs*, values of timers - used by a module to check whether messages are received from others - bear no relationship with timeliness constraints (the $d(\theta)$'s in our case).

Hence, termination (ADeliveries and completion of tasks in Θ in our case) occurs within time latencies that are necessarily greater than achievable lower time bounds. In other words, as observed previously, the resulting timeliness-centric FCs are worse (further away from necessary and sufficient conditions) than FCs obtained when considering a synchronous solution. Timeliness is stronger than liveness.

Let Ω_1 be assertion "*accuracy and completeness properties hold whenever needed*". Let Ω_2 be assertion "*valuations of FDs' timers ensure specified timeliness property* | *accuracy and completeness properties hold*". Examples of issues raised are as follows:

(1) Estimate coverage(Ω_1),
(2) Estimate ϕ_1 = coverage(*RAs under which* Ω_1 *holds*),
(3) Estimate coverage(Ω_2),
(4) Estimate ϕ_2 = coverage(*RAs under which* Ω_2 *holds*),
(5) Check that ϕ_1 and ϕ_2 are at least equal to $C(S, V(s))$.

A problem is that doing (1) or (3) accurately enough is incredibly difficult, if not unfeasible. This is the reason why it is necessary to consider immersion in a synchronous system. Assume for a while that synchronous assumptions are never violated.

Firstly, *FDs* used by asynchronous solutions become perfect detectors (strong completeness and strong accuracy). Therefore, any weaker properties (e.g., strong completeness and eventual weak accuracy for $\diamond S$) certainly hold. Hence, issue (1) vanishes.

Secondly, *FDs'* timer values can be set equal to the Net delay upper bound. FCs - embedded in a dimensioning oracle - can then be run in order to check

whether every (valued) task deadline is met. Whenever the oracle returns a "yes", the specified timeliness property holds for sure. Consequently, issue (3) vanishes.

This is clearly not the case when values chosen for *FDs'* timers are guessed optimistically. Timers values chosen being smaller than the actual upper bound, the timeliness property and/or its valuations are violated, despite positive response from the oracle. If timers values are guessed pessimistically (values greater than the upper bound), then the oracle turns pessimistic as well, i.e. $V(\langle rtuab \rangle)$ which is feasible is declared unfeasible.

Synchronous Solutions These solutions are based on synchronized rounds, whereby messages sent during a round are received during that round, or on bounded message delays; see [16] for examples.

Synchronous solutions can be defeated by timing failures. A timing failure is a violation of proven (under assumptions), or postulated, lower (early timing failure) and upper (late timing failure) bounds on delays.

There are two approaches for addressing issues raised with timing failures in synchronous models. One consists in transforming *algorithms* proved correct in the absence of timing failures [16]. The other consists in transforming the *models*, by imposing restrictions on timing failures. Both have related coverages. We explore timing failures in the sequel.

Comments No matter which design solution is considered, we have to show how to correctly implement synchronous assumptions, i.e. how to address issues raised with timing failures, so as to arrive at residual assumptions that have computable and "very high" coverages.

For the sake of conciseness, we will only focus on how to correctly design, dimension and implement a synchronous Net module, and under which conditions a synchronous Net in the presence of timing failures is not equivalent to an asynchronous Net.

However, let us sketch out how, going through a dimensioning phase, a lower bound on each processor speed - some of the variables in $[s]$ - can be determined, $V([s])$ being needed to implement synchronous (processing) modules. There are two possibilities with respect to variables $x(\theta)$, which are expressed in "code length" by S/W engineers.

Either only variables $x(\theta)$ are provided as inputs for conducting a dimensioning phase. Then, FCs and deadlines $d(\theta)$ - valued by a client - are used to compute an upper bound on acceptable worst-case execution time for every task θ, noted $wcet(\theta)$, such that the valued problem under consideration is declared feasible.

Or, in addition to variables $x(\theta)$, upper bounds $wcet(\theta)$ are also provided by S/W engineers, considering that each task runs alone over its module. Then, FCs serve to tell whether the valued problem under consideration is feasible.

In both cases, the ratio $x(\theta)/wcet(\theta)$ is the lowest acceptable processor speed for task θ [4]. Knowing the mapping of set Θ onto set Q, it is then trivial to compute the speed lower bound for each of the modules [5].

5 Timing Failures

Let b and B be, respectively, the targeted lower and upper bounds of delays for transmitting messages through Net, with $B > b$.

Under our assumption that implementation of $\{[S], V([s])\}$ is faultless, there are two causes for timing failures, namely overloads and excessive failure densities. One easy approach to these issues consists in pretending or assuming rarity (timing failures do not occur "too often"). Typical residual assumptions are as follows:

(RA): it is always the case that every message sent or broadcast by a module is received in at most B time units "sufficiently often" or by a "sufficiently high" number of correct modules.

There are synchronous algorithms that work correctly in the presence of "small" densities of timing failures. They simply ignore late messages. Hence, rigorously speaking, this approach boils down to avoiding the problems raised with timing failures. However, estimating coverage(RA) is hardly feasible.

How often, how long is Net overloaded or impacted by "too many" internal failures? Such phenomena must be under control.

For the sake of conciseness, we will not explore the issue of excessive densities of failures (other than timing failures). Let us briefly recall that any design correctness proof rests on "up-to-f-out-of-n" assumptions (see f/W in $\langle RTUAB \rangle$, which is the assumption that, in W time units, no more than f modules in Q - i.e. among n modules - may stop). Valuations consist in assigning values to variables f only, matching values of variables n (lower bounds) deriving from feasibility conditions.

It is worth noting that, unless proof-based SE principles are obeyed, it is impossible to assert that f cannot become equal to n. This is due to the fact that any SE fault made while conducting a design phase and/or a dimensioning phase is inevitably replicated over the n modules. Whenever this fault is activated, all n modules fail (at about the same time). This is what has happened with the maiden flight of European satellite launcher Ariane 5 [12].

SE faults being avoided, causes of violations of proven bound B are violations of, (i) valuations of variables f (covered by $C(X, V(x))$, (ii) residual assumptions.

Let us now discuss ways of addressing the issue of overload prevention. Then, we will examine how to cope with timing failures via detection.

[4] In either case, some of the acceptable bounds $wcet(\theta)$ may be very small, a result of having chosen very small task deadlines or very high external event arrival densities. This could translate into lowest acceptable processor speeds too high to be affordable.

[5] This does not raise scheduling problems. These have necessarily been solved during a design phase, otherwise FCs would not be available.

5.1 Prevention

Overloads should not be viewed as uncontrollable phenomena or as a result of necessarily fuzzy engineering tradeoffs. Module Net has a specification $\langle Net \rangle$, which must be provably derived from pair $\{\langle RTUAB \rangle$, solution $\mathcal{A}\}$. For instance, $\langle m.Net \rangle$ should state message arrival models provably derived from the tasks and the external event arrival models stated in $\langle RTUAB \rangle$, from $\mathcal{A}$, and from the analysis of the task schedules produced by $\mathcal{A}$.

With some $\mathcal{A}$s - e.g., periodic algorithms - it is possible to show that message arrivals obey a unimodal arbitrary model. A difficulty stems from the need to establish tight bounds on arrival densities, i.e. ratios $a(message)/w(message)$ for every *message*. Then, it is possible to *prove* that, considering some Net architecture and some message scheduling algorithm (a protocol), message delays are bounded by B.

In fact, it is possible to do even better. For instance, in [7], considering multiaccess broadcast channels (off-the-shelf technology) and tree protocols, one shows how to guarantee delay upper bounds on an individual message basis, i.e. a bound $B(message)$ proper to every *message*. Establishing bound(s) b usually derives easily from establishing bound(s) B.

Many $\langle Net \rangle$-like problems involving periodic or sporadic message arrivals have well known solutions (see [19] for examples).

Proven bound(s) B may be violated if bounds on message arrival densities are violated, i.e. with a probability no greater than 1 - $C(X, V(x))$. Indeed, recall that under a proof-based SE approach, such bounds are (exactly or pessimistically) derived from bounds $a(t)/w(t)$ stated in $\langle RTUAB \rangle$.

It follows that coverage (*message arrival densities*) $\geq$ coverage (*external event arrival densities*) $\geq C(X, V(x))$.

Despite the fact that, by its very nature, $C(X, V(x))$ cannot be equal to 1, one can nevertheless enforce bound B, by exercising external admission control (see [14] for example), which amounts to perform on-line model checking. On each Net entry point, incoming message μ is rejected whenever it is found that bound $a(\mu)/w(\mu)$ is violated [6]. In this case, bound B cannot be violated, which comes at the expense of having some messages experience infinite waiting, with a probability at most equal to 1 - $C(X, V(x))$.

Consequently, it is possible to *prove* that the delay experienced by every message carried by Net has B as an upper bound, this holding true whenever residual assumptions are not violated. Example of a residual assumption:

(RA): speed of physical signals over man-made passive communication links is 2/3 of light speed at least.

To summarize, SE faults being avoided, a correct specification $\langle Net \rangle$ can be established. A bound B can be proven to hold, and coverage(*proven bound B cannot be violated*) $\simeq 1$, given the RAs usually considered.

[6] As for arrival models, one can also perform external on-line model checking for failure models, and "reject" (e.g., stop) a module that would violate its failure semantics.

5.2 Detection

It is granted that timing failures can occur, whatever the causes, albeit not "too often". An approach, suggested by many authors, aims at transforming timing failures into omission failures: A message whose delay does not range between b and B is not "seen" by algorithm $\mathcal{A}$. This implies the existence of a "filtering" algorithm, noted $\mathcal{F}$, sitting in between Net and $\mathcal{A}$. A Net timing failure detected by $\mathcal{F}$ running on module q is transformed into a module q receive omission. There are many synchronous algorithms that work correctly in the presence of bounded omission failures.

Can there be algorithms $\mathcal{F}$ that detect violations of bounds b and B? If not, which bounds can be enforced with certainty? Under which residual assumptions? Densities of "fake" omission failures - i.e., transformed timing failures - must be "under control". How?

The 1W solution Let us review a popular solution, which will be referred to as the 1-way (1W) solution. It is based on the requirement that every module owns a clock (n clocks in Q) and that clocks are ϵ-synchronized [7], that is:

for any two clocks c_i and c_j, for any (universal) observation time T,

$$| c_i(T) - c_j(T) | < \varepsilon, \quad \varepsilon > 0.$$

Correct clocks are necessary to achieve ϵ-synchronization. A correct clock is a clock whose deviation from physical time is negligible over short time intervals, e.g. intervals in the order of small multiples of B. Clocks must resynchronize more or less regularly, via remote clock readings (message passing), to achieve ϵ-synchronization. This can be done by having each module equipped with a GPS (Global Positioning System) or a radio receiver. In this case, very small values of ϵ can be attained [8]. However, it might not be realistic to assume that each module in a critical system is physically located so that there is an unobstructed line-of-sight path from its antenna to GPS space vehicles. Or ϵ-synchronization must be maintained despite receivers going down or being jammed. In the sequel, we assume that a few modules - called time servers - are equipped with such receivers, and that system-wide ϵ-synchronization is achieved via some algorithm whereby modules remotely read any of these time servers via messages sent through Net (or read their clocks, mutually, whenever no time server is up).

Every message is timestamped with sender's local time (at send time), as well as with a receiver's local time (upon receipt). The difference is the message delay, as observed by a receiver, noted *od*. Timeliness tests retained for $\mathcal{F}$ are applied by receivers. $\mathcal{F} = 1W/test$ consists in rejecting only those messages that fail *test*.

A message whose real delay - noted *rd* - lies within interval $[b, B]$ will be said to be correct. Otherwise, a message will be said to be incorrect.

[7] ϵ-synchronization is a well known Approximate Agreement problem.

[8] see [18] for a comprehensive presentation of recent results.

Let $u = B - b$. Real lower bound is β, $0 < \beta \leq b$. There is no upper bound for a late timing failure. With critical problems such as $\langle RTUAB \rangle$, early timing failures matter as much as late timing failures. Indeed, in order to establish timeliness-centric FCs, one must identify worst-case scenarios, which is achieved by resorting to adversary proofs and arguments, whereby messages controlled by the "adversary" experience smallest delays, while messages issued by the task under consideration experience highest delays. Any violation of delay lower bounds would invalidate the proofs that underlie the FCs.

With ε-synchronized clocks, for some given od, real delays range between $od - \varepsilon$ and $od + \varepsilon$. Therefore, stricto sensu, the following restrictive timeliness test RTT should be considered:

$$RTT\colon b + \varepsilon \leq od \leq B - \varepsilon.$$

A problem is that every message, including correct messages, is rejected with $\mathcal{F} = 1W/RTT$. Indeed, it has been shown that ε has $\varepsilon_0 = u(n-1)/n$ as a lower bound with deterministic clock synchronization algorithms, assuming perfect (i.e. non drifting) clocks [15]. With correct clocks, it follows that $\varepsilon = \gamma \varepsilon_0$, $\gamma > 1$.

It can be easily verified that $B - \varepsilon$ cannot be greater than $b + \varepsilon$. Hence, RTT does not make sense.

One could then consider semi-restrictive timeliness tests $SRTT$s, such as:

$$SRTT\colon b + \alpha u/2 \leq od \leq B - \alpha u/2, \quad 0 < \alpha < 1.$$

With such tests, a correct message whose real delay is δ is accepted only if transmitted while sender's clock is less than $\delta - (b + \alpha u/2)$ ahead of receiver's clock and sender's clock is less than $B - (\alpha u/2 + \delta)$ behind receiver's clock.

Such tests are seldom considered, given that it is generally felt unacceptable to reject correct messages possibly arbitrarily often. With such tests, the initial issue of estimating the coverage of a bounded density assumption relative to timing failures translates into the issue of estimating the coverage of a bounded density prediction relative to accepted correct messages!

Consequently, only permissive timeliness tests are considered. The classical permissive test, PTT, suggested by many authors, is as follows:

$$PTT\colon b - \varepsilon \leq od \leq B + \varepsilon.$$

Of course, every correct message is certainly accepted. However, incorrect messages may also be accepted, whose real delays satisfy any of the following constraints:

$$max\{\beta, b - 2\varepsilon\} \leq rd < b, \qquad B < rd \leq B + 2\varepsilon.$$

Surprisingly, this appears to be barely addressed in the literature.

In summary:

(1) Assuming ε-synchronized clocks, it is impossible to detect violations of bounds $b - \varepsilon$ or $B + \varepsilon$ with certainty when using the $1W$ solution.

(2) Only early timing failures such that $rd < b - 2\varepsilon$ and late timing failures such that $rd > B + 2\varepsilon$ are certainly detected with $\mathcal{F} = 1W/PTT$.

(3) Bound on message delay uncertainty - initially postulated to be u - can only be amplified with $\mathcal{F} = 1W/PTT$. Actual bound on message delay uncertainty U guaranteed to some algorithm $\mathcal{A}$ by $1W/PTT$ is:

$$U = u[1 + 2\gamma(1 - 1/n)] + (b - \beta), \quad \text{if } \beta > b - 2\varepsilon$$
$$U = u[1 + 4\gamma(1 - 1/n)], \quad \text{if } \beta \leq b - 2\varepsilon.$$

(4) With $\mathcal{F} = 1W/PTT$, bounds on tasks response times or, equivalently, feasibility conditions for having the timeliness property stated in $\langle RTUAB \rangle$ hold true, must be established considering that the actual upper bound on Net message delays is $B' = B + 2\varepsilon$, rather than B.

(5) With any filtering algorithm $\mathcal{F}$ based on $1W$ that would use a test more permissive than PTT, the difficult issue of estimating the coverages of the following assertions is raised:
– every timing failure is detected,
– every correct message is accepted.

The ε-synchronization assumption Given that we are assuming timing failures, it is very much appropriate to examine whether those conditions under which ε-synchronization is achieved are or are not invalidated by incorrect messages, when considering clock synchronization algorithms based on message passing. Unfortunately, it is impossible to use $1W$ so as to enforce bounds b and B on delays of clock synchronization messages - which is required to enforce ε - given that restrictive test RTT cannot be contemplated.

Solutions other than $1W$ exist. However, they come at the expense of some amplification of the achievable ε lower bound. In synchronous systems subject to timing failures, ε-synchronization either rests on the *assumption* that synchronization messages exchanged among clocks are correct, and then "small" values of ε can be contemplated, or ε-synchronization holds despite timing failures, provided that "greater" values of ε are acceptable.

Let us explore the coverages involved with ε-synchronization, and compute the "cost" of establishing "guarantees" for one simple example.

Coverage that ε holds true, denoted $C(\varepsilon)$, is an increasing function of $\varepsilon = \gamma\varepsilon_0$. Picking up low values for γ leads to low coverages. Indeed, for $\gamma \simeq 1$, $C(\varepsilon) \simeq 0$. Is there a value ε^* yielding $C(\varepsilon) \simeq 1$? The answer is yes. We sketch out below an example of an algorithm $\mathcal{F}$ that permits to safely detect timing failures impacting clock synchronization messages. This algorithm, denoted $2W/TT$, has previously been proposed for solving the remote clock reading problem [3]. For the sake of simplicity, we neglect factors in time derivatives (which is equivalent to assuming perfect clocks).

Timing failures are detected by each module, by locally measuring message round-trip delays (2-ways, hence 2W). A module q that triggers such a measure, by issuing a request message at time T_1, is to receive a response message res from the module being polled, referred to as the sender. Without loss of generality,

assume that it takes no time for the sender to generate *res*. Let T_2 be the time at which *res* is received by q. Variable *rd* is the real delay of message *res*. $T_2 - T_1$ is the (observed, i.e. real) round-trip delay.

Timeliness test TT applied by q is:

$$TT\colon 2b \leq T_2 - T_1 \leq 2B.$$

Hence, early timing failures experienced by *res* that are not detected with certainty are such that $\beta \leq rd < b$, whereas late timing failures experienced by *res* that are not detected with certainty are such that $B < rd \leq 2B - \beta$.

Consequently, clock synchronization message delay uncertainty is $2(B - \beta)$. It follows that $\varepsilon^* = 2(u + b - \beta)(n - 1)/n$.

Knowing a lower bound of guaranteed ε when using $\mathcal{F} = 2W/TT$, it is trivial to re-compute what can be achieved with $\mathcal{F} = 1W/PTT$ w.r.t. messages issued by tasks, as well as to quantify the "cost" incurred with achieving certainty (i.e. $C(\varepsilon) \simeq 1$). Let us examine the "cost" - denoted ΔB - incurred with guaranteeing a message delay upper bound.

$B'_0 = B + 2\varepsilon_0$ being the highest upper bound that cannot be considered ($\gamma = 1$), it follows that $\Delta B = B + 2\varepsilon^* - B'_0 = 2[u + 2(b - \beta)](n - 1)/n$.

This "cost" has $2\varepsilon_0$ as a lower bound, attained if one makes no mistake in predicting lower bound b.

Note that checking timing failures with $2W$ increases the message load offered to Net. Bounds b and B with $2W$ would then be greater than the bounds considered without $2W$. However, no matter which message-based clock synchronization algorithm is considered, it is possible to resort to piggybacking: Clock synchronization data is transmitted taking advantage of "normal" messages (e.g., those generated by $\mathcal{A}$, by system S/W or application S/W components). In this case, neither bound b nor bound B is affected by clock synchronization traffic.

In summary:

(6) It is possible to achieve ε-synchronization in the presence of timing failures, considering residual assumptions that have coverages higher than usually asserted. Indeed, given that ε-synchronization is known to be achievable in synchronous systems in the presence of a minority f of faulty clocks (e.g., $n > 3f$ with f byzantine clocks), it follows that we have moved from (RA): clocks are ε-synchronized, to (RA): $n - f$ clocks at least are correct. The latter being only one of the assumptions that underlie the former, its coverage is necessarily higher.

(7) Synchronous systems in the presence of timing failures are not necessarily equivalent to asynchronous systems.

(8) With $\mathcal{F} = 2W/TT$ for clock synchronization messages, $\mathcal{F} = 1W/PTT$ for task messages, and under (RA): $n - f$ clocks at least are correct, the actual delay upper bound that is guaranteed - with some coverage $\geq \phi$ - for messages exchanged among modules that own correct clocks is $B^* = B + 4\varepsilon^*$, which has $B + 4\varepsilon_0$ as a lower bound. Claims or proofs of quantified "real-time behavior" based on considering that $B + \varepsilon$ is the actual upper bound are flawed.

Many clock synchronization algorithms could have been considered. Other algorithms $\mathcal{F}$ exist, and other bounds and coverages can be established.

Nevertheless, apart from numerical calculations, these conclusions have general applicability.

5.3 Prevention or Detection?

It is worth noting that there is no choice. It is very risky to resort to timing failure detection only, as there would be no way of showing that a "good" bound B has been picked up, no matter which timeliness test is considered [9]. Therefore, coverage(*no correct message is rejected*) has no significance. The concept of a "correct" message becomes elusive when permissive timeliness test PTT turns out to be unduly restrictive in fact, i.e. bearing no relationship with reality. Would real B be greater than guessed B, proven liveness or timeliness properties would not hold at run-time.

For instance, ε-synchronization is achievable provided that clock synchronization messages are tested correct "often enough" (e.g., once per period, for periodic clock synchronization algorithms). Similarly, algorithmic solutions would serve no real purpose if their messages - as well as those generated by tasks - are not tested correct "often enough".

Consequently, timing failure prevention is mandatory for developing solutions such that coverage(*no correct message is rejected*) and coverage(*every incorrect message is rejected*) are, (i) accurately estimated, (ii) "very high".

For any such problem as $\langle Net \rangle$, which is correctly derived from some higher level problem by applying proof-based SE principles, there exists a matching bound B, which must and can be *established*, exactly or pessimistically, by solving a distributed real-time scheduling problem, as briefly explained under Section 5.1. A proven bound B, noted $\mathcal{B}$, being established, the definition of a *correct* message matches reality. Therefore, messages tested incorrect are incorrect indeed. As a result, it is possible to show that the following coverages can be very close to 1, given the RAs that can be considered:

- coverage(*no message whose real delay is $\mathcal{B}$ at most is rejected*),
- coverage(*every message whose real delay is greater than $\mathcal{B}^*$ is rejected*).

Note also that doing this is the only way to bound densities of "fake" omission failures (with sufficiently high coverages).

With critical systems, there is definitely no choice. Irrespective of which computational models are contemplated for a design phase, synchronous system models must be considered for dimensioning and implementation phases.

With synchronous models, predictability assorted with "very high" coverages is illusory, unless (off-line) prevention and (on-line) detection of timing failures

[9] Bounds B estimated via statistical analysis of traffic measurements have poor coverages most often. Furthermore, predictability obligations also apply to systems that are yet to be built.

are resorted to jointly. Hence, the relevance of proof-based SE principles for such systems [10].

6 Conclusions

We have examined issues that arise with critical applications and systems, for each of the four phases that span a project lifecycle, from initial problem capture, to system implementation, when conducted according to proof-based system engineering principles. In particular, assuming correct implementations of H/W and S/W components, we have explored those irreducible sources of uncertainty which restrict predictability, as well as the concept of coverage applied to problems, solutions, assumptions.

A generic problem that arises with critical applications such as, e.g., air traffic control/management, namely the real-time uniform atomic broadcast problem, has been presented. Two design styles, namely asynchronous and synchronous solutions, have been compared in terms of resulting assumptions as well as coverages. This has led us to revisit popular justifications for asynchronous models, as well as popular approaches to the construction of synchronous systems, paying special attention to the issues of overloads and timing failures.

Critical systems raise many open research issues. Examples of areas where further work is needed are formal methods for proof-based system engineering, models and properties class structuring, algorithms for solving combined real-time, distributed, fault-tolerant computing problems, complexity and combinatorial analysis for establishing necessary and sufficient feasibility conditions.

¿From a more general perspective, for generic problems raised with critical systems, it would be very useful to compare design and dimensioning solutions, feasibility conditions, and associated coverages, as obtained under, respectively, deterministic and stochastic approaches.

Acknowledgements

I would like to thank Anders P. Ravn for his comments and suggestions that helped improving the paper.

References

1. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Asynchronous Systems, Journal of the ACM 43(2) (March 1996) 225-267.
2. Chandra, T.D., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus, 12th ACM Symposium on Principles of Distributed Computing (August 1992) 147-158.

[10] Specifications such as $\langle Net \rangle$ or bounds B are almost never *established* under current SE practice. Therefore, overloads occur at unpredictable rates. It follows that, as stated in the Introduction Section, dominant causes of such overloads are SE faults, rather than S/W or H/W faults.

3. Cristian, F.: Probabilistic Clock Synchronization, Distributed Computing (3) (1989) 146-158.
4. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the Presence of Partial Synchrony, Journal of the ACM, 35(2) (April 1988) 288-323.
5. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process, Journal of the ACM 32(2) (April 1985) 374-382.
6. Hadzilacos, V., Toueg, S.: A Modular Approach to Fault-Tolerant Broadcasts and Related Problems, Technical Report TR 94-1425, Cornell University (May 1994), 83 p.
7. 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 (May 1998) 360 - 369.
8. Kuhn, D.R.: Sources of Failure in the Public Switched Telephone Network, IEEE Computer (April 1997) 31 - 36.
9. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem, ACM Trans. on Programming Languages and Systems 4(3) (July 1982) 382 - 401.
10. Le Lann, G.: Proof-Based System Engineering and Embedded Systems, in Embedded Systems, Springer-Verlag LNCS on Embedded Systems (G. Rozenberg, F. Vaandrager Eds.) (to appear in 1998) 41 p.
11. Le Lann, G.: On Real-Time and Non Real-Time Distributed Computing, invited paper, 9th Intl. Workshop on Distributed Algorithms, Springer-Verlag LNCS 972 (J.M. Hélary, M. Raynal Eds.) (1995) 51-70.
12. 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 (March 1997) 339 - 346.
13. Leveson, N.G., Turner, C.: An Investigation of the Therac-25 Accidents, IEEE Computer (July 1993) 18 - 41.
14. Liebeherr, J., Wrege, D.E., Ferrari, D.: Exact Admission Control for Networks with a Bounded Delay Service, IEEE/ACM Trans. on Networking, 4(6) (December 1996) 885-901.
15. Lundelius, J., Lynch, N.A.: An Upper and Lower Bound for Clock Synchronization, Information and Control 62(2-3) (August-September 1984) 190 - 204.
16. Lynch, N.A.: Distributed Algorithms, Morgan Kaufmann Pub., ISBN 1-55860-348-4 (1996) 872 p.
17. Powell, D.: Failure Mode Assumptions and Assumption Coverage, 22nd IEEE Intl. Symposium on Fault-Tolerant Computing (July 1992) 386-395.
18. Special Issue on Global Time in Large Scale Distributed Real-Time Systems, Schmid, U. Guest Editor, Journal of Real-Time Systems 12(1-2) (1997) 230 p.
19. Tindell, K., Burns, A., Wellings, A.J.: Analysis of Hard Real-Time Communications, Journal of Real-Time Systems 9(2) (1995) 147-171.