Asynchrony and Real-Time Dependable Computing

Gérard Le Lann, INRIA, B.P. 105, 78153 Le Chesnay Cedex, France
Gerard.Le_Lann@inria.fr

Abstract

We examine how computer system problems can be derived from real application problems, with a particular focus on the relevance of some assumptions, especially those related to computational models. Then, we compare these models, ranging from pure synchronous to pure asynchronous semantics, to conclude that synchrony does not necessarily dominate asynchrony whenever one is concerned with real operational systems. The issue as to whether asynchronous solutions can be considered for designing and building real-time distributed dependable systems is addressed. A priori, time free solutions are antagonistic with proving timeliness properties. We show how to circumvent this apparent contradiction via the late binding principle. This principle, as well as drawbacks of synchronous solutions, are illustrated.

1 Introduction

Most *real* computer-based systems are bound to meet safety and/or liveness and/or dependability and/or timeliness requirements, to some degree. For example, most *real* applications specify timeliness constraints, be they "hard" real-time constraints (critical applications) or performance objectives (non critical applications). Consequently, a number of open theoretical questions are worth investigating from a practical perspective as well. We examine the following question: When *designing a real-time system*, meant to be used for some *real application*, is it mandatory to consider a synchronous computational model?

This issue seems to be highly controversial, for obscure reasons in some cases, for obvious non scientific reasons in other cases. We believe that any useful contribution to this long lasting controversy should be based on an understanding of how *real* application specifications are established, as well as on how a *real* system specification can be proved correct vis-à-vis some given *real* application specification.

In Section 2, we elaborate on an essential system engineering activity, namely the "application requirements capture" phase, which raises complex theoretical and practical issues. This serves to explain why a designer – a computer scientist, an engineer – is not entitled to "simplify" the specification of a user's problem. For any given pair {problem, solution}, specified logical properties and QoS should be guaranteed to hold true with some coverage, which depends on a number of assumptions, computational models in particular. In Sections 3 and 4, merits of various computational models are discussed, as well as the superiority of asynchronous solutions for some problems in distributed fault-tolerant computing. The late binding principle is presented and illustrated in Section 5. Related work is briefly examined in Section 6.

Our results permit to conclude that, for some applications, real-time distributed systems built out of asynchronous solutions are safer and more efficient than systems based upon (partially) synchronous solutions. Whether similar results can be established for other applications is an open – and interesting – question.

2 Real application problems and computer system problems

Let $\langle A^* \rangle$ denote the exact (ideal) specification of some *real* application problem, and let $\langle A \rangle$ denote some *real* specification of that same problem, arrived at after conducting a "requirements capture" phase. A solution, i.e. a computer-based system, denoted S, is looked for.

As is the case with every (theoretical, practical) problem, $\langle A \rangle$ is the union of two specifications, one that stipulates axiomatics/assumptions, another that stipulates properties sought (to be demonstrated to hold under stated assumptions). Therefore, $\langle A \rangle$ writes $\{\langle m.A \rangle; \langle p.A \rangle\}$, where $\langle m.A \rangle$ stipulates *problem models*, i.e. environmental and technological assumptions regarding future S, and $\langle p.A \rangle$ stipulates those services and QoS that S should deliver (*problem properties*). $\langle A \rangle$ also specifies cov($\langle p.A \rangle$), a lower bound set for the coverage of $\langle p.A \rangle$ under $\langle m.A \rangle$. The coverage of an assertion ϕ - denoted C(ϕ) – is the probability or the likelihood that this assertion holds true [16].

Of course, $\langle A \rangle$ can only be expressed in some natural language. For example, in air traffic control, $\langle p.A \rangle$ would include "The discrepancy between the exact coordinates of

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

an airplane and the coordinates seen by an air traffic controller should never exceed so much", $\langle m.A \rangle$ would include "Radio communications may go down from time to time", $\text{cov}(\langle p.A \rangle) > 1 - 10^{-7}$.

Given that "requirements capture" involves human interventions as well as tools built by humans, that phase cannot perfect. Hence, $\langle A \rangle$ can only be an approximation of $\langle A^* \rangle$. Usually, $\langle p.A \rangle$ matches $\langle p.A^* \rangle$ quite well – users know what they want. Conversely, $\langle m.A \rangle$ may differ from $\langle m.A^* \rangle$ to a certain extent. Humans cannot predict the future accurately. For example, $\langle m.A \rangle$ stipulates some "worst-case" radiation level, which is later revealed to be lower than real levels.

Clearly, the smaller the "distance" – denoted α, measured on a scale (0, 1) – between $\langle A \rangle$ and ideal $\langle A^* \rangle$, the better in terms of future user's satisfaction. Without trying to be formal, let us consider that the coverage of a specification $\langle A \rangle$ is equal to 1 - α. Consequently, "simplifying" a user's problem is not a very good idea, for this amounts to diminish the coverage of initial specification $\langle A \rangle$.

Various analyses of project failures and accidents with computer-based systems have identified faulty requirements capture as being the major cause of failures and catastrophes, rather than "software faults". A relatively well documented case is the failure of Ariane 5 flight 501 [17].[1]

$\langle A \rangle$ being informal, it is mandatory to translate $\langle A \rangle$ into $\langle X \rangle$, a specification of a matching computer system problem (e.g., computer science, computer and electrical engineering) so that proof techniques can be applied. Given $\langle X \rangle$, one can undertake design activities, and demonstrate whether [Δ], i.e. system S specification, satisfies $\langle X \rangle$. If $\langle X \rangle$ is a correct translation of $\langle A \rangle$, and [Δ] satisfies $\langle X \rangle$, then [Δ] is a correct solution for $\langle A \rangle$ – which cannot be established directly.

Subsequent phases, which consist in proving that [Δ] is correctly implemented in hardware, software or any convenient technology, are of no concern here.

In order to eliminate most likely faults, it is recommended that $\langle A \rangle$ and $\langle X \rangle$ be constructed simultaneously. Set $\langle m.X \rangle$, which is the translation of $\langle m.A \rangle$, specifies the future operational "adversary" of system S, i.e. event arrival models ("loads"), process models, failure models and so on. Obviously, every model in $\langle m.X \rangle$ must have a coverage at least equal to $\text{cov}(\langle p.A \rangle)$. Set $\langle p.X \rangle$ specifies such properties as, e.g., 1-copy data serializability, uniform atomic broadcast, strict termination deadlines. For our purposes, properties of interest in $\langle p.X \rangle$ are safety, liveness, and timeliness ("real-time"), denoted SafeP, LiveP, and TimeP, respectively. Dependability properties are "encapsulated" within SafeP, LiveP and TimeP, which properties should hold for failure models and failure occurrence models specified in $\langle m.X \rangle$.

[1] Ariane is the name of the European satellite launcher.

Rather than trying to translate informal $\langle A \rangle$ directly into a formal specification $\langle X \rangle$ (such attempts have always failed), we have devised a less ambitious, albeit rigorous, goal. Only those terms that have formal definitions may appear in sets $\langle m.X \rangle$ and $\langle p.X \rangle$. Proof obligations are unambiguous, and can be fulfilled. These principles underlie TRDF, a proof-based system engineering method that we have developed and used to conduct projects with a number of partners since 1995 [13]. Whenever some COTS product is selected a priori, TRDF is particularly helpful in showing whether such a choice is or is not incompatible with $\langle m.X \rangle$. In a particular instance (air traffic control, with DGAC – French FAA), an impossibility result [7] helped us in catching this kind of mistake early enough.

Obviously, the smaller the "distance" – denoted β – between $\langle A \rangle$ and $\langle X \rangle$, the better in terms of future user's satisfaction. Without trying to be formal, let us consider that the coverage of a specification $\langle X \rangle$ varies proportionally to $(1 - \beta)(1 - \alpha)$. Consequently, "simplifying" a computer system problem so as to facilitate system design and validation work is not a very good idea, for this amounts to diminish the coverage of initial specification $\langle X \rangle$.

Examples of simplifications can be given. In $\langle m.X \rangle$, the failure model for a processor is stated to be "not worse" than the omission model, and an event arrival model is specified to be periodic, period p, when in fact, real $\langle m.X \rangle$ should state a Byzantine failure model, and a "periodically sporadic" event arrival model, with some sporadicity parameters smaller than p. Real $\langle X \rangle$ being "falsified", future system S cannot behave as desired, even if it has been proven that (1) [Δ] correctly satisfies (simplified) $\langle X \rangle$, (2) S correctly implements [Δ]. This is so for the reason that the future adversary of S will be "stronger" than postulated as per (simplified) $\langle m.X \rangle$.

A common instance of an arbitrary simplification is the (systematic) choice of a synchronous computational model, stated in $\langle m.X \rangle$, when some "less simple" model should in fact be selected. There are many real applications which are poorly modeled with synchrony assumptions. Examples are applications involving mobile users, wireless communications, applications meant to tolerate intrusions (e.g., denial of service), to name a few.

Too often, users are led to make unfounded (and risky) choices, under the influence of designers, who know that it is easier to solve problems in a synchronous model. Even when synchrony assumptions are founded – low level cyclic kinds of computations – cost considerations may lead to conclude that asynchronous solutions should be favored.

It may also happen that no computational model is stated in $\langle m.X \rangle$, in which case the choice of a model is under the responsibility of a designer. However, such a choice is constrained by $\text{cov}(\langle p.A \rangle)$ – see ((R1), (R2)) in Section 3.1.

When conducting theoretical work, any computational

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

model may be considered. Theoretical works are essential, in that they permit to establish optimality and/or impossibility results. However, when considering *real* systems, that is when issues raised with (1) implementing a postulated model, (2) assessing the coverage of an implemented model, must be addressed, then the choice of a computational model is constrained by a number of requirements.

Let *Proofs* be the set of demonstrations proving that $[\Delta]$ solves $\langle X \rangle$ under some *design assumptions*. $[\Delta]$ should be such that the specification of desired properties $\langle p.X \rangle$ cannot be violated as long as (1) the adversary behaves as specified by $\langle m.X \rangle$, (2) *design assumptions* are not violated.

Trivially, the simpler "falsified" $\langle X \rangle$, the easier the establishment of *Proofs*. Unfortunately, such *Proofs* are of no interest. Being established for "falsified" $\langle X \rangle$, their coverage is arbitrarily poor. This explains many operational "difficulties" encountered with systems based upon static synchronous designs and/or built out of software components coded via some "synchronous" language. Resorting to a "time-triggered" approach or to some "synchronous programming language" a priori and systematically, without questioning the appropriateness of such choices, are examples of implicit arbitrary – hence risky – simplifications.

3 On computational models and coverage

For our purposes, it suffices to consider that a design solution Δ consists of a set of (virtual or physical) software and hardware modules, structured after some architecture, and equipped with a set of algorithms that govern their collective behavior. We restrict our scope of attention to deterministic algorithms. System S that implements $[\Delta]$ is structured after a number of levels of abstraction or implementation, ranging from 1 for the lowest level (basic hardware) to z, the highest level encompassed by $[\Delta]$, i.e. the level of specification $\langle X \rangle$. The discussions that follow apply to models encompassing levels up to z.

3.1 The computational model spectrum

Let $\mathcal{M}(\Delta)$ stand for a model considered at design time. $\mathcal{M}(\Delta)$ may be either stated in $\langle m.X \rangle$ or chosen when starting a design phase.

Computational models range from pure synchronous – denoted Sync, to pure asynchronous – denoted pure Async, characterized by their intrinsic *timing assumptions* [15]. Consider, in $[\Delta]$ and *Proofs*, variables that represent *timing assumptions* for computational or communication steps, for every level of interest. As defined in [5] and [6], pure synchrony means that every such variable is assumed to be finite and bounded, (upper, lower) bounds being known at design time, while pure asynchrony means that any such variable may be infinite. As will be seen, asynchrony means that every such variable is assumed to be finite and unbounded. By definition, coverage C($\mathcal{M}(\Delta)$) is the probability or likelihood that none of those timings postulated via $\mathcal{M}(\Delta)$ can be violated at runtime.

The choice of $\mathcal{M}(\Delta)$, when stated in $\langle m.X \rangle$, is bound to meet the following coverage requirements (obvious):

- (R1): C($\mathcal{M}(\Delta)$) can be accurately computed,
- (R2): C($\mathcal{M}(\Delta)$) > cov($\langle p.A \rangle$).

When the choice of $\mathcal{M}(\Delta)$ is left to a designer, for a correct design to be an acceptable solution, it must be that C(*design assumptions*) > cov($\langle p.A \rangle$). $\mathcal{M}(\Delta)$ being an element of *design assumptions*, the choice of $\mathcal{M}(\Delta)$ must meet coverage requirements ((R1), (R2)) as well.

A non-existing assumption cannot be violated. Hence, coverage issues involved with ((R1), (R2)) do not arise with the pure Async model. Unfortunately, many problems of interest in dependable distributed computing do not have deterministic solutions in this model [7]. This has motivated work directed at "augmenting" this model with some semantics, so as to circumvent impossibility results. One can identify two classes of "added semantics", namely timed semantics and time-free semantics. Unreliable Failure Detectors [3] are an example of time-free semantics.

Models that match the pure Async model augmented with timed semantics are known under the name of partially synchronous models [5], [6]. Within this set, we consider models where some modules and/or some levels are assumed to match pure synchrony assumptions whereas others match pure asynchrony assumptions.[2] They will be denoted ParSync. In the Sync model, every module or level matches pure synchrony assumptions. Sync being a particular case of ParSync models, every result established for ParSync models applies to Sync a fortiori. Models that match the pure Async model augmented with time-free semantics will be referred to as asynchronous models – denoted Async. Let $l_{\mathcal{M}(\Delta)}$ stand for the highest level encompassed by *timing assumptions* proper to $\mathcal{M}(\Delta)$. By definition: $l_{Async} = 0$ and $l_{ParSync} > 0$.

3.2 Implemented computational models

Let us now consider system S, to be built out of basic technology assumed to exhibit *basic timing properties* that ought to be fully trusted (no coverage issue, by assumption). It follows that *basic timing properties* are restricted to be plain physical hardware timing properties, such as processor instruction-level timings or bit propagation delays over a communication link.

Let s^* stand for the highest level encompassed by *basic timing properties*. Ideally, $s^* = 1$. Any *timing assump-*

[2] Partially synchronous models where postulated timing bounds hold true only after some unknown future time do not match real-time semantics.

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

tion stated for some level higher than s^* has some coverage, bound to meet requirements ((R1), (R2)).

Let $l^*_{ParSync}$ (resp., l^*_{Async}) stand for the highest level concerned with a provably correct implementation of a ParSync (resp., an Async) model. With ParSync, trivially, $l^*_{ParSync} = l_{ParSync}$. With Async, although $l_{Async} = 0$, $l^*_{Async} > 0$, given that some synchrony must be assumed in order to implement the time-free semantics of interest.

Typically, $l^*_{ParSync}$ is the application or middleware level (z or "close to" z) whereas l^*_{Async} is some low level communication protocol level. In [10], one gives an implementation of *Strong* and *Perfect* Failure Detectors [3] – denoted FDs – that makes lists of suspected/crashed processors available at a level denoted COM, which can be the UDP/TCP protocol level in general-purpose systems, or the physical link protocol level in special-purpose systems. For example, Fast FDs [10] can be implemented at the link protocol level in spaceborne systems [2]. Processes are provided with those (time-free) safety and liveness properties that define FDs, which makes it possible to use pure asynchronous algorithmic solutions at level l^*_{Async} and above. With ParSync, it is argued that processes are provided with some timed properties, which makes it possible to use pure synchronous algorithmic solutions at level $l^*_{ParSync}$ and above.

Recall that time-free semantics of interest are those strictly necessary for circumventing impossibility results in distributed fault-tolerant computing. They are "low level" semantics. Therefore:

(F1) $\quad l^*_{ParSync} > l^*_{Async}$.

Let us now consider *timing assumptions* relative to level $(s^* + 1)$. There are two possibilities.

One is that values of variables involved with these *timing assumptions* are "guessed", in which case it is impossible to compute their coverage. Requirements ((R1),(R2)) cannot be met.[3] Such "guesses" underlie the TA model [4] and the TCB approach [18].

The other possibility – the correct one – consists in conducting a worst-case schedulability analysis – denoted wcs analysis – for level $(s^* + 1)$ processes, assuming level s^* *basic timing properties*, which yields computable analytic **predicted** *worst-case timeliness* bounds, which are proven *timing assumptions* valid for level $(s^* + 1)$. And so on, until reaching level $l^*_{ParSync}$ or l^*_{Async}.

Let T^k stand for level k **predicted** *worst-case timeliness* bounds. In Section 4.1, we recall what is involved with a wcs analysis, which must be conducted whenever X is a real-time problem. What we have established at this point is the following:

• Even if X is not a real-time computing problem, one must conduct wcs analyses whenever a ParSync model has been selected, so as to prove its implementation correctness.

• Any (correct) implementation of a ParSync model is problem X-dependent. Indeed, $\forall k$, $s^* < k \leq l^*_{ParSync}$, bounds T^k must derive from wcs analyses conducted for the entire set of models (processes, data, event arrivals, failures, etc.) identified at level k, which necessarily depends on the set of models specified in $\langle m.X \rangle$.

Given the three possible outcomes of a wcs analysis – see Section 4.1 – one can conclude as follows:

• In the absence of wcs analyses, the coverage of a ParSync design is unknown.

• In the presence of wcs analyses, with a ParSync design,
- SafeP, LiveP and TimeP may have a "poor" coverage,
- real performance/efficiency may be "poor".

• It is always the case that C(Async) $\geq$ C(ParSync).

• Some problems have optimal Async solutions that dominate optimal ParSync solutions, in terms of achieved performance and/or efficiency.

4 TimeP, SafeP and LiveP

4.1 Timeliness

Consider real-time computing problem X – TimeP appear in $\langle p.X \rangle$. Any (proven) solution comprises, for every level k, from z to $(s^* + 1)$ – see scheduling theory:

• some solution $[\Delta]$ based upon scheduling algorithms, such as e.g., HPF, EDF [14] or more complex schemes, traditionally designed in some ParSync model,

• timeliness proofs derived from worst-case ("loads" and failures) scheduling analyses, valid for level k derivation of pair $\{\langle X \rangle, [\Delta]\}$, conducted considering some ParSync model, necessarily,

• computable analytical expressions of *worst-case timeliness* bounds T^k (bounds T^z matching those specified as per TimeP), such as, e.g. termination deadlines, as well as feasibility conditions valid for pair $\{\langle X \rangle, [\Delta]\}$.

Ability to predict values of bounds T before S is turned on is an absolute requirement with real-time systems. Unfortunately, as is well known, wcs analyses are involved (NP-complete problems, most often), despite the fact that, inevitably, they rest on simplified models of (1) COTS products (e.g., processors, system-level software), (2) internal system architectures, (3) external and internal event arrivals ("loads"), (4) tasks, (5) faulty behaviors. Moreover, in order to bring the inherent complexity of such analyses down to some tractable level, approximations are often resorted to.

[3] The conventional counter-argument is as follows: Let us pick up "very large" timing values, so that they are "almost never" violated. This is flawed, for the reason that without a schedulability analysis, it is impossible to tell whether or not congestion or thrashing may occur at level $(s^* + 1)$ or higher. If the case, real timing values are infinite, which ruins any expectation regarding the achievement of some decent coverage.

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

Although sound from a mathematical viewpoint, such approximations add to the inaccuracy due to considering simplified models of technology, of reality.

A worst-case execution time (denoted *wcet*) must be predicted for every process under consideration. A *wcet* depends on a number of system parameters and technological choices (e.g., processor hardware types and speed), not always accessible with sufficient accuracy. Moreover, finding a *wcet* for a process that is distributed or replicated across processors necessarily involves a wcs analysis. For example, worst-cases for message delays cannot be computed or measured assuming that every process runs "alone".

Let Θ^k stand for **real** *worst-case timeliness* bounds experienced at run-time at level k and θ^k stand for **real** *delays*. By definition, θ^k might be significantly smaller than Θ^k.

As is well known, there are three possible outcomes for a wcs analysis:

(Ω_1) "optimistic" analysis, i.e. $T^k < \Theta^k$, hence bounds T^k may be violated at run-time,

(Ω_2) exact analysis, i.e. feasibility conditions are necessary and sufficient, hence $T^k = \Theta^k$,

(Ω_3) "pessimistic" analysis, i.e. $T^k > \Theta^k$, hence bounds T^k are overdimensioned.

Consider some level $k \geq l^*_{Async}$. A ParSync design makes explicit use of bounds T^k via timers, watchdogs, clocks, and so on. Bounds T^k being "wired in", the temporal behavior of a ParSync design is entirely determined by the "quality" of wcs analyses. Which is not the case with Async designs, where bounds T^k serve solely to predict worst-case temporal behaviors. They are not integrated into a design.

In case of outcome (Ω_1), all properties SafeP, LiveP, TimeP are lost with a ParSync design. Moreover, as pointed out previously, at every level, wcs analyses depend fully on problem X under consideration. Conversely, with an Async model, the design of an implementation of time-free semantics does not depend on problem X. Hence, wcs analyses needed to prove Async implementation correctness can be conducted ignoring X fully or quasi-fully. In [10], the only dependencies are due to the non-preemptive nature of some resources. For example, one must know the maximum length of a message at the physical link level. Consequently:

(F2) The complexity of a wcs analysis is (significantly) smaller with an Async model.

Also, recall that wcs analyses encompass every level up to $l^*_{ParSync}$ with a ParSync model whereas they encompass every level up to l^*_{Async} with an Async model.

Given **(F2)** and **(F1)**, the probability of outcome (Ω_1) is smaller with Async. Hence the conclusion:

(C1) C(Async) $\geq$ C(ParSync).

Note that **(C1)** rewrites C(Async) $>$ C(ParSync) whenever COTS products are used to implement levels above l^*_{Async}. This is due to the fact that the $[\Delta_{COTS}]$ internal to a COTS product is never disclosed with sufficient accuracy. Furthermore, no COTS product is accompanied with TimeP proved for some models. Hence, it is very difficult – if not unfeasible – to conduct a wcs analysis that would be valid for pair $\{\langle X \rangle, [\Delta]\}$. Hence, requirements ((R1), (R2)) are almost never met whenever a ParSync model is to be implemented using COTS products.

Given **(F2)** and **(F1)**, the probablity of outcome (Ω_2) is higher with Async. See [10] for an example, where exact bounds are established for FD-message transmission delays. It is in general highly unlikely that exact bounds can be established for "high level" delays, that fully depend on some problem X. Regarding performance/efficiency figures, the observations made for outcome (Ω_3) – see below – apply also for outcome (Ω_2), given that worst-case scenarios do not occur too often in general.

In case of outcome (Ω_3), bounds T^k being "wired in" ParSync solutions, these solutions are necessarily slower and/or less efficient than Async solutions, whenever solutions are of comparable logical complexity – the case for, e.g., Uniform Consensus (see further).

A typical example is the use of bounds for end-to-end transmission delays experienced by interprocess messages, whenever desired orderings (of events, of state transitions) must be enforced. Let T^k (resp., t^k) stand for the predicted worst-case upper (resp., best-case lower) bound. If global time is assumed – precision ϵ – a transmission delay can be measured by a receiver, with precision ϵ. Consider a message that has travelled in real θ^k, measured τ^k. A receiving process should wait $T^k - \tau^k + \epsilon$ before making an ordering decision, which entails a $T^k + 2\epsilon$ worst-case latency. If "good" non synchronized clocks are assumed (no global time), a receiving process should wait $T^k - t^k$ before making an ordering decision, which entails a $\Theta^k + T^k - t^k$ worst-case latency.

Such latencies are not experienced with Async solutions. Indeed, an Async solution "works" at speeds determined by θ^k most of the time, by Θ^k under worst-case scenarios. In our example, with an Async solution, ordering decisions can be made whenever some logical (i.e. time-free) condition is met, which may occur any time between θ^k and Θ^k, entailing a Θ^k worst-case latency. Hence the conclusion:

(C2) When coverage is not an issue, Async solutions may be faster and/or more efficient than ParSync solutions.

Novel solutions for the Uniform Consensus problem [10] illustrate **(C2)**. We conjecture that similar results can be established for other problems in distributed real-time dependable computing.

Note that conclusions **(C1)** and **(C2)** are in accordance with observations made previously by many researchers.[4]

[4]Excerpt from [15]: "It is impossible or inefficient to implement the synchronous model in many types of distributed systems".

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

4.2 Safety and liveness

Assume X is not a real-time computing problem – SafeP and LiveP only appear in $\langle p.X \rangle$. From **(C1)**, for $k > s^*$, it follows trivially that $C(\text{SafeP}^k_{Async}) > C(\text{SafeP}^k_{ParSync})$ and $C(\text{LiveP}^k_{Async}) > C(\text{LiveP}^k_{ParSync})$.

Let us illustrate the above with SafeP. Many asynchronous algorithms that preserve SafeP regardless of $\langle m.X \rangle$ being or not being violated have been published. Moreover, Async algorithms may preserve SafeP despite violations of $\mathcal{M}(\Delta)$, i.e. violations of time-free semantics "added" to the pure Async model. Such algorithms are called "indulgent" in [8]. An example with Uniform Consensus is the $\Diamond S$ rotating coordinator algorithm of [3], which preserves SafeP even if $\Diamond S$ FD semantics are violated. Consequently, for some problems, SafeP hold true *under no conditions* with Async algorithms, not the case with ParSync algorithms.

Then the question: Given that, whenever X is not a real-time computing problem, choosing an Async model maximizes C(SafeP) and C(LiveP), why is it that ParSync models are considered when SafeP and LiveP only must be demonstrated?[5]

Conclusions **(C1)** and **(C2)** are the foundations of the late binding principle.

5 The late binding principle

Let X be a real-time computing problem ("hard" real-time or specific "performance" properties) and S be a real-time computing system proved correct vis-à-vis X. How can TimeP be proved with Async designs? An essential observation is as follows: *In a project life-cycle, it is necessary to consider some ParSync model only when time has come to conduct schedulability analyses*.

Let $\mathcal{M}(S)$ be the implementation model that matches system S. That $\mathcal{M}(S)$ might be some Sync or ParSync model does not imply that $\mathcal{M}(\Delta)$ has to be a Sync or ParSync model as well.

Δ may well be designed in some Async model. Indeed, wcs analyses for asynchronous designs are feasible – contrary to assertions repeatedly stated in some circles.

The idea of deferring the consideration of some $\mathcal{M}(S)$ until after having devised and proved some design Δ in some $\mathcal{M}(\Delta)$ less "restrictive" than $\mathcal{M}(S)$ has been stated first in [12], echoed in [9] and [11], and detailed in [13], under the name of "design immersion" (in a computational model). This is equivalent to the concept of *late binding* (of a design to some computational model), a well known concept in the areas of programming languages and compilation.

[5] Again, theoretical works are not concerned with this question.

According to this principle, bounds T^k_{Async} are established **only after** SafeP^k_{Async} and LiveP^k_{Async} have been proven, which is done without assuming any bounds T^j_{Async}, $j \leq k$. This has definite advantages (see Sections 3 and 4), that are inaccessible to those approaches based upon "early binding" to a ParSync model, where one must first establish bounds $T^{k-1}_{ParSync}$ – or, even worse, postulate such bounds – prior to proving $\text{SafeP}^k_{ParSync}$ and $\text{LiveP}^k_{ParSync}$.

With this principle, the apparent contradiction between, (1) retaining Async solutions when designing real-time systems and (2) the need to consider some ParSync model to conduct wcs analyses for proving TimeP, vanishes.

The late binding principle consists of the following three steps, step 1 preceding steps 2 and 3. Steps 2 and 3 may be concurrent.

• Step 1: Given $\langle A \rangle$ or $\langle X \rangle$, select as $\mathcal{M}(\Delta)$ the most appropriate Async model that meets the cov($\langle p.A \rangle$) constraint. Then, specify Δ (selecting asynchronous algorithms only) with time-free predicates stating activation conditions for schedulers. For example: "Service waiting queue W whenever W is non empty", or "Make local scheduling decisions whenever distributed consensus has been reached". Prove SafeP and LiveP.

• Step 2: Design a solution for implementing the time-free semantics of $\mathcal{M}(\Delta)$ out of level s^* *basic timing properties*. Conduct a wcs analysis in the ParSync model that matches $\mathcal{M}(S)$ and provide computable *timeliness* bounds proper to that solution (e.g., failure detection latency).

• Step 3: Do a "late binding" of Δ to the ParSync model that matches $\mathcal{M}(S)$, so as to conduct wcs analyses yielding computable *timeliness* bounds T for pair $\{\langle X \rangle, [\Delta]\}$.

We have applied the late binding principle while revisiting the Uniform Consensus (UC) problem. In [10], $\mathcal{M}(\Delta)$ is the pure Async model augmented with *Strong* or *Perfect* FDs. The FD "immersion" process (in a Sync model) has led us to introduce Fast FDs, i.e. FDs that achieve *computable* failure detection times d that are worst-case optimal, i.e. small compared to worst-case interprocess message transmission delays D. This has resulted into novel algorithms that have optimal *wcet*'s – denoted Z below.

With $FastUC$, a novel Async solution for UC, Z_{Async} is sublinear in $f\,D$, f standing for the maximum number of processor crashes. Under conditions easily met most often with real systems, $FastUC$ achieves $Z_{Async} = D$ (the absolute lower bound).

This lower bound is inaccessible to solutions proven optimal in the pure Sync model that do not take advantage of Fast FDs. Indeed, by definition, given that $f > 0$, $Z_{Sync} \geq 2\,D$.

In [1], Fast FDs are considered along with the Sync model. Novel optimal *wcet*'s have been established. For example, for UC, we have shown that $Z_{Sync} = D + f\,d$.

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

6 Related work

Examples of ParSync models are the Timed Asynchronous Distributed System (TA) model [4] – where "asynchrony" means "pure Sync with mistakes" – and the Timely Computing Base (TCB) approach [18]. TA and TCB rest explicitly on *timing assumptions*, that are an integral part of designs/solutions – "early binding" to a Sync model. Given that there is **no difference between TA or TCB and the pure Sync model**, TA and TCB are uninteresting from a theoretical viewpoint (nothing new). They are uninteresting also from a practical perspective. It is acknowledged that "guessed" *timing assumptions* may be violated. One would then expect to see convincing solutions such that requirements ((R1), R2)) – see Section 3.1 – are met. Unfortunately, such solutions are not given.

With TA, the very difficult issues raised with computing the coverage of pure Sync assumptions are believed to be addressed satisfactorily simply by stating that "A TA system alternates between *good* and *bad* periods", "Violations of "guessed" timings occur infrequently", which are meaningless assertions in the absence of wcs analyses. Assume that $\text{cov}(\langle p.A \rangle) = 1 - 10^{-4}$, a very modest requirement. How can it be "believed" that every "good" period will last 3,599.64 seconds at least every hour? It is suggested that the coverage of pure Sync assumptions can be derived from measurements performed on pre-existing systems. This is flawed for obvious reasons. Problems X addressed with the TA model are not real-time computing problems. Hence, the question asked end of Section 4.2 arises with TA.

Proponents of TCB allude to the need for conducting wcs analyses, but fail to address it. Rather, they resort to the very classical – mistaken – "receipe": Find some "good" hardware that does away with the problems. What is called a "control system/network" is supposed to implement the pure Sync components central to TCB, so that *timing assumptions* are (magically) transformed into "guaranteed" timed services, made accessible to other components such as, e.g. application-level processes. Obviously, varying – possibly "high" – loads are experienced by a "control system/network". Loads within a TCB system are problem X-dependent, those developing at the exposed interface of a "control system/network" and within it in particular. Hence the question: How can requirements ((R1), (R2)) be met without conducting problem X-dependent wcs analyses? Basically, there is no difference between TCB and what has always been known under the name of a "real-time distributed" computing system.

Knowing that violations of optimistic level k "guessed" *timing assumptions* lead to violations of SafeP and LiveP (in addition to TimeP) at level k, proponents of TA and TCB have suggested that *timing assumptions* could be "enforced" via on-line detection of "performance failures", by supplementing design Δ with measure-compare-and-kill algorithms that serve to (1) timestamp every significant state transition, (2) measure every actual delay value, for every level k delay variable that appears in Δ or *Proofs*, (3) compare every measured delay with its postulated bound. In case a "performance failure" occurs, that failure is transformed into a provoked abort, – discard a "late" incoming message, abort a "late" process, crash a processor that has received a "late" message or performed a "late" computation. In [10], it is explained why "performance failures" may go undetected arbitrarily often. Hence, SafeP, LiveP (in addition to TimeP) may be violated arbitrarily often. Whenever a "performance failure" is detected, a TA or a TCB system turns mute, which may also happen arbitrarily often. Who could be interested in using such systems?

The pure Sync model is a particular case of ParSync. With pure Sync, one considers variable delays. The static Sync model – denoted StatSync – is a particular case of pure Sync, where delays are assumed to be constant. This assumption is not questionable when considering levels up to level s^*. For any other level, this assumption serves to "hide" contention and/or waiting queue phenomena that arise from resource/data sharing. Examples of use of StatSync are the "time-triggered" (TT) approach and the semantics that underlie many "synchronous programming languages" (Statecharts, Esterel, to name a few).

Contrary to intuition, variability cannot be avoided with real systems and environments, even for the "simplest" cases. Real systems comprise a number of software components, every component having an execution time determined by a number of parameters, which themselves may depend on external operational conditions. Hence variability. Software components are usually multiplexed over processors.[6] Consequently, they must be scheduled, which introduces variations in their actual start times and termination times. StatSync designs such as TT designs may be very inefficient/slow, for they consist in devising a cyclic time frame of fixed duration, decomposed into a fixed number of time slots, each slot having some fixed duration and being assigned to a given process. Anything new? Of course not. This type of designs has been known for more than 30 years under the name of SSTDM (static synchronous time division multiplexing). Obviously, the inefficiency of such designs is proportional to the "silence ratio" in time slots. In fact, as is well known, SSTDM leads to designs that are dimensioned so as to accommodate simultaneous individual (per processor) worst-case scenarios – a max $\{\max\}$ function – which inevitably results into (possibly significantly) overdimensioned and/or slow systems, given that, most often, some individual worst-case scenarios are mutually exclusive – a min $\{\max\}$ function suffices.

[6] If not the case, then so-called "distributed" systems are plain juxtapositions of independent processors – the case with many TT systems.

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

7 Conclusions

The late binding principle seems to be an interesting novel concept that reconciles real-time computing and asynchrony. This opens up new territories to researchers and practitionners. The superiority of asynchronous computational models and solutions in terms of coverage are believed to be generic, i.e. valid for every *real* problem/system. Results regarding the superiority of asynchronous designs/solutions in terms of efficiency or performance have been established for some problems. Whether similar results can be established for other problems is an open – and interesting – issue. Cross-fertilization of various disciplines – e.g., scheduling theory, distributed algorithms – should lead to new fundamental results, helping to advance theory and practice in areas of crucial importance.

Acknowledgements The author would like to thank Carole Delporte and Hugues Fauconnier for stimulating discussions about this work.

References

[1] M. K. Aguilera, G. Le Lann and S. Toueg, "On the Impact of Fast Failure Detectors on Real-Time Fault-Tolerant Systems", *Proc. of the 16th Intl. Workshop on Distributed Algorithms (DISC)*, Springer-Verlag LNCS 2508, Oct. 2002, 354-369.

[2] Project A3M on Advanced Avionics Architecture and Modules, funded by ESTEC (European Space Agency), software developments by ASTRIUM and AXLOG Ingenierie, 2001-2003 (reports available from ESTEC and ASTRIUM).

[3] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, 43(2), March 1996, 225-267.

[4] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model", *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999, 642-657.

[5] D. Dolev, C. Dwork, L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus", *Journal of the ACM*, 34(1), Jan. 1987, 77-97.

[6] C. Dwork, N. Lynch, L. Stockmeyer, "Consensus in the Presence of Partial Synchrony", *Journal of the ACM*, 35(2), April 1988, 288-323.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus With One Faulty Process", *Journal of the ACM*, 32(2), April 1985, 374-382.

[8] R. Guerraoui, "Indulgent Algorithms", *Proc. of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000, 289-297.

[9] R. Guerraoui and A. Schiper, "Consensus: the Big Misunderstanding", *Proc. of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*, Oct. 1997, 183-188.

[10] J.-F. Hermant and G. Le Lann, "Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems", *IEEE Transactions on Computers*, 51(8), Special Section on Asynchronous Real-Time Distributed Systems, Aug. 2002, 931-944.

[11] M. Hurfin and M. Raynal, "Asynchronous Protocols to Meet Real-Time Constraints: Is It Really Sensible? How to Proceed?", *Proc. of the IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, April 1998, 290-297.

[12] G. Le Lann, "On Real-Time and Non Real-Time Distributed Computing", invited paper, *Proc. of the 9th Intl. Workshop on Distributed Algorithms (DISC)*, Springer-Verlag LNCS 972, Sept. 1995, 51-70.

[13] G. Le Lann, "Proof-Based System Engineering and Embedded Systems", invited paper, *Proc. of the European School on Embedded Systems*, Nov. 1996, Springer-Verlag LNCS 1494, Oct. 1998, 208-248.

[14] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, 20(1), Jan. 1973, 46-61.

[15] N. A. Lynch, "Distributed Algorithms", ISBN 1-55860-348-4, Morgan Kaufmann Pub., 1996, 872 p.

[16] D. Powell, "Failure Mode Assumptions and Assumption Coverage", *Proc. of the 22nd IEEE Intl. Symposium on Fault-Tolerant Computing*, July 1992, 386-395.

[17] Safety Critical Mailing List, Department of Computer Science, University of York, http://www.cs.york.ac.uk/hise/hise4/frames9.html, Archived Contributions, "Contribution on the Failure of Ariane 5 flight 501", posted March 1999.

[18] P. Verissimo, A. Casimiro, C. Fetzer, "The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness", *Proc. of the IEEE Intl. Conference on Distributed Systems and Networks*, July 2000, 533-542.

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE