# Time-Utility Scheduling and Provably Correct Critical Computer-Based Systems

Gérard Le Lann
*INRIA, Rocquencourt, France*
*Gerard.Le_Lann@inria.fr*

## Abstract

*This paper investigates ways of expanding the scope of applicability of time-utility and aggregate utility driven scheduling. Being interested in critical applications and systems, we explore issues raised with proving that a system is endowed with combined safety, liveness, timeliness and dependability properties, the province of proof-based system engineering. We examine the nature of proof obligations, as well as how to fulfill them, whenever timeliness and aggregate utility properties are sought. Relationships with classical real-time computing problems and timeliness proofs are analyzed. Then we take time-utility scheduling a few steps further, by showing how to maximize aggregate utility while achieving process serializability, process termination, as well as dependability properties, in various computational models, considering distributed systems prone to failures where processes share multicopied updatable persistent data.*

## 1. Introduction

Timeliness properties – denoted TimeP – and their proofs are essential with real-time systems. Time-Utility Functions (TUF's) and Aggregate Utility (AU) optimization criteria [1, 2, 3, 4, 5] are generalizations of classical – and more restrictive – timeliness attributes and properties commonly considered in real-time computing, such as strict process termination deadlines. Throughout this paper, TimeP achieved by resorting to TUF-driven scheduling algorithms aimed at maximizing AU are denoted TU-TimeP.

In addition to TimeP or TU-TimeP, most computer-based systems (CBS's) must exhibit properties belonging to the three classes of safety, liveness and dependability – denoted SafeP, LiveP and DepP, respectively. A number of intricate issues arise with the design of CBS's that are bound to exhibit some combination of properties belonging to these four classes. A typical example arises (frequently) with CBS's where updatable persistent data/variables are shared by (application-level, system-level) processes. A SafeP that is mandatory with such systems is "data consistency", more formally defined as "process serializability" [6]. Issues raised with achieving TU-TimeP and process serializability altogether have not been addressed so far. Current work concentrates on achieving TU-TimeP and resource-level mutual exclusion altogether [5]. Resource-level mutual exclusion does not imply process serializability.

In Sections 4 and 5, we investigate ways of expanding the scope of applicability of TUF/AU scheduling, by circumventing underlying restrictions, and focusing on algorithmic issues. We show how TUF scheduling may work in any kind of CBS, i.e. in distributed systems of replicated or non replicated processors and data, where AU must be maximized *system-wide*, in the presence of shared updatable persistent data and partial failures, for various models of computation (beyond synchronous models).

Furthermore, we examine the (more difficult) proof issues. In the case of life/mission/business-critical CBS's, one must be able to predict the future behavior of a to-be-deployed CBS, with very high accuracy and confidence. Given the complexity of current and future application-level services and properties, of current and future computing and networking technology, it is quite clear that existing system design approaches have reached their limits: they do not permit predictions with high enough accuracy or confidence. Following the historical pattern of more mature engineering disciplines (e.g., electronics, telecommunications), system engineering (SE) for CBS's has reached an inflexion point in its history. Time has come for replacing ad hoc or empirical techniques with more "scientific" techniques, notably proof-based techniques, such as those at the core of a novel methodological trend, known under the name of Proof-Based System Engineering (PBSE), which is sketched out in Section 2. PBSE and its relationships with TUF/UA scheduling are explored in Section 3.

## 2. Proof-Based System Engineering for CBS's

The first phase in a lifecycle is the application-centric Requirements Capture (RC) phase. According to published statistics (from NASA, in particular), this is the phase where most faults are made, revealed (years) later, usually through unitary or integration testing, a very costly way of finding out that "something is wrong". Under a PBSE approach, an RC phase is conducted until the application of interest is fully and unambiguously *specified* as a problem expressed as computer-based requirements (CBR's).

In an attempt to minimize or eliminate ambiguities and inconsistency, the formal software engineering (FSWE) community has advocated for formally expressed CBR's, e.g. in languages based upon temporal logic. Without tangible success, apparently. For good reasons. Human


IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

beings in general, CBS users and designers in particular, do not "speak" formal languages. Furthermore, semantics that are tractable with existing FSWE methods constitute a limited subset of those semantics which faithfully represent real world CBS's technology and operational environments, real world user/application requirements. With existing FSWE methods and tools, formal specifications can be considered only in the late phases of a lifecycle. Lastly, there is more than software involved with a CBS [7], and FSWE can only follow PBSE in a lifecycle [8, 9].

PBSE is based on a different approach. A CBR is not a formal specification. Rather, a CBR is expressed in some natural language (English, French, German, etc.), with restrictions: a term (other than article, conjunction, etc.) may appear in a CBR only if it has a formal definition in some scientific discipline – see Figure 1 for a simplified example. This permits mutual understanding between the various stakeholders involved in an RC phase. Specifications that derive from a CBR in subsequent lifecycle phases get increasingly formal. This is how PBSE bridges the gap between reality and what is tractable with current FSWE methods and tools.

A CBR comprises two subsets, one denoted <p.CBR>, which serves to specify which services and properties ought to be delivered/guaranteed by a CBS, another one, denoted <m.CBR>, which serves to specify the (future) operational CBS adversary, i.e. those models and assumptions under which one must prove that <p.CBR> is not violated. Inevitably, <p.CBR> states some combination of SafeP, LiveP, TimeP and DepP. Classical examples are process atomicity or mutual exclusion for SafeP, eventual process termination for LiveP, deadline-constrained termination for TimeP, uniform consensus or high availability for DepP.

A CBS is a usable, implemented, solution for a given CBR. A CBS is an implementation of a design specification, denoted S. Under a PBSE approach, it is required to prove that S solves CBR. The output of an SDV (System Design & Validation) phase, which has a CBR as an entry, is specification S, along with proofs showing that S satisfies CBR. As a result, any CBS that faithfully implements "blueprint" S is (provably) endowed with properties at least as "strong" as those specified in <p.CBR>, in the presence of an adversary at least as "aggressive" as specified in <m.CBR>. In other words, such a CBS will always "win against" its specified adversary.

Meeting proof obligations at CBR/CBS levels, i.e. in early lifecycle phases, is at the core of a novel PBSE method – the TRDF method [8, 9] – pioneered by INRIA since 1995, and assessed with European partners in such various domains as, e.g., Integrated Modular Avionics, Nuclear Power Plants, Satellites, Air Traffic Control, Complex Systems of Systems. PBSE is believed to be extremely efficient at reducing costs, delays and risks of projects/systems failures by significant ratios.

***<m.CBR>***

- distributed processors, replicated processors and data
- application process models: finite graphs, assignment over processors is unrestricted, wcet's and inter-process causal dependencies are known
- processes read/write shared persistent data, no restriction
- set of processes, set of shared data items, are open-ended
- processor failure models: stop, omission.
- process activation models: sporadic, aperiodic
- processor failure occurrence model: aperiodic
- computational model: synchronous.

***<p.CBR>***

- SafeP: process serializability, process atomicity (other safety properties are "encapsulated" in DepP)
- LiveP: every process whose activation has been requested eventually terminates
- TimeP: for every process, a strict termination deadline (from a dozen of milliseconds to one second) to be met
- DepP: failure detection within bounded latency, atomic broadcast, atomic commit.

**Figure 1. Excerpts from an IMA problem (military avionics) addressed with the PBSE/TRDF method [10] (French MoD, Dassault Aviation, INRIA)**

Note that with a PBSE method, it is possible to make a clear distinction between, (1) proving that "blueprint" S is correct vis-à-vis some given CBR (an SDV activity) and, (2) proving that S is correctly implemented (a development activity). Making such a distinction is essential. Indeed, it does not help much to prove that some software or hardware component correctly implements some given specification – possibly following some formal software or hardware engineering method – if it has not been checked beforehand that this specification is a correct one (for the problem considered).

The TRDF method is the foundation of the PBSE work to be conducted within ASSERT, a 3-year long Integrated Project launched by the European Commission in 2004, led by the European Space Agency, involving 32 partners (national agencies, companies, research laboratories).

SafeP and LiveP proofs are proofs in Logic, essentially. Most often, TimeP proofs are proofs in various kinds of Analytical or Algebraic Calculi, in Combinatorial Optimization.[1] Note that LiveP proofs must be established before proving TimeP. Indeed, proving TimeP involves

[1] Hence, contrary to widespread belief in certain circles, TimeP proofs are not reducible to SafeP proofs.



0-7695-2132-0/04/$17.00 (C) 2004 IEEE

conducting some worst-case schedulability analysis, so as to establish a set of constraints known as timeliness feasibility conditions (FC's). These analyses and related FC's, which are based upon individual process worst-case execution times (wcet's) or exact execution times, are valid only if it is proved that processes terminate. DepP proofs are SafeP proofs, LiveP proofs, as well as proofs in various kinds of Stochastic Calculi.

## 3. PBSE and TUF/AU Scheduling

With TU-TimeP, the nature of proof obligations does not differ much from that of proof obligations to be fulfilled with classical TimeP. Let $\Omega$ stand for the on-line scheduler specified in "blueprint" S.

### 3.1. TU-TimeP Proof Obligations

Classical TimeP proofs derive from timeliness FC's, which result from conducting worst-case schedulability analyses for pair {<m.CBR>, S}, viewing $\Omega$ as "playing against" <m.CBR>. First, an analytical upper bound R(k) on response times must be established for every process k. Assuming no overloads, FC's for "hard" real-time CBS's consist in writing:

$\forall k,\ R(k) < D(k)$, D(k) standing for process k's strict termination deadline (specified in <p.CBR> for application-level processes, derived from <p.CBR> for system-level processes).

It is the duty of system designers, with the help of an SDV tool, to express timeliness FC's first. When this set of computable analytical constraints is available, it can be entered into another PBSE tool, referred to as a Feasibility & Dimensioning (FD) Oracle. An FD Oracle can be used as often as needed, assigning desired numerical values to CBR variables, in order to check whether there exists a valuation of variables in S that meets the chosen dimensioning of CBR variables. The output of an FD Oracle is either "yes" (TimeP holds true for every process) or "no" (TimeP is lost for some process(es)).

Similarly, regarding TU-TimeP proofs, one establishes an analytical lower bound U(k) for the utility achieved by every process k, by conducting worst-case schedulability analyses, based upon exact processes execution times (see Section 4.2). FC's consist in writing:

$$\forall T,\ \forall k \in K(T),\ \Sigma_T U(k)/\Sigma_T U°(k) > \alpha,\ 0 < \alpha < 1,$$

where T stands for any time interval, K(T) is the set of processes that may terminate within interval T, and U°(k) stands for the highest utility associated with process k (specified in <p.CBR> for application-level processes, derived from <p.CBR> for system-level processes).

Observe that specifying $\alpha$ is equivalent to setting a lower bound for the density of process termination deadlines to be met. SDV tools and FD Oracles can/should be used. An FD Oracle serves to check whether some valuation of $\alpha$ ($\alpha$ stated in <p.CBR>) is or is not achieved.

In both cases (TimeP and TU-TimeP), NP-hard or NP-complete combinatorial optimization problems are to be addressed, on two grounds, namely with on-line schedulers and with FC's. This is clearly the case with the IMA problem shown in Fig. 1, due to (a) the process/processor assignment problem, (b) the serializability property, in the presence of sporadic and aperiodic process activation laws, i.e. arbitrarily interleaved executions of processes – in addition to TimeP. As for TU-TimeP, a proof of NP-hardness can be found in [11]. The R(k)'s and U(k)'s depend on two parameters. One is the distance of scheduler $\Omega$ from optimality. Another is the distance of FC's from optimality, i.e. necessary and sufficient FC's. Theoretical optimal bounds, denoted $R_{\Downarrow}(k)$'s and $U_{\Uparrow}(k)$'s, are obtained only if both distances are null.

### 3.2. Schedulers and Feasibility Conditions

Every scheduling algorithm has some intrinsic complexity, which translates into some execution duration, denoted $X_\Omega$. Picking up a scheduler shown to be optimal on theoretical grounds is not always appropriate. The closer to (theoretical) optimality, the higher $X_\Omega$. Also, the choice of activation laws for an on-line scheduler is driven by the magnitude of $X_\Omega$. Activating a scheduler upon every process activation request may not be appropriate. Optimal bounds of relevance – denoted R*(k) and U*(k) – are the $R_{\Downarrow}(k)$'s or the $U_{\Uparrow}(k)$'s where $X_\Omega$ and the worst-case scheduler activation latencies – denoted $\{L_\Omega\}$, since there might be more than one such latency – are accounted for. Consequently, it is often the case that a theoretically sub-optimal scheduler should be retained. Obviously:

$$\forall k,\quad R_{\Downarrow}(k) < R^*(k) < R(k) \quad \text{and} \quad U(k) < U^*(k) < U_{\Uparrow}(k).$$

The (simplified) generic optimization problems are:

- Find scheduler $\Omega$ and $\{L_\Omega\}$ that minimize the R*(k)'s – for classical TimeP,
- Find scheduler $\Omega$ and $\{L_\Omega\}$ that maximize the U*(k)'s – for TU-TimeP.

With non "extreme" TUF's (see Figure 2), scheduling algorithms of weak polynomial complexity (in the order of $n^2 \log n$ or $n^3$, for n processes to be scheduled) can be close enough to optimality, which optimality may be identified by resorting to, e.g., dynamic programming [11].

In the case of "extreme" TUF's, i.e. arbitrarily shaped functions, possibly varying with time, results published so far derive from simulation work or measurements on


IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

prototypes. Analytical work has been started recently [5] for specific CBR's (see Section 4.2). Regarding scheduling algorithms, analyses and proofs of global AU optimization, it might be advisable to "import" results established in such disciplines as Decision Theory or Game Theory. For example, efficient TUF/UA schedulers might be derived from strategies devised in Game Theory, companion theorems being the proofs to be entered into an SDV tool.

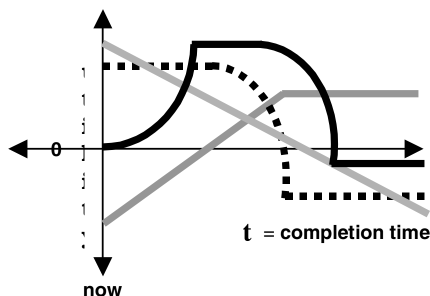


**Figure 2. Four example time/utility functions (excerpted from [4])**

A notion, common to Game Theory and Distributed Algorithms, which deserves some special attention is that of knowledge. In centralized or uni-processor systems, or when considering strictly local (processor-wise) AU optimization in distributed systems, there is exactly 1 "decider" involved, the unique or local scheduler, that knows the "players" (the processes) and their individual "moves". Hence, in such systems, knowledge (every decider knows) is a natural assumption. In distributed systems where global AU optimization is required (the $\alpha$ ratio), multiple "deciders" are involved. Therefore, one has to decide which way to go:

- Make scheduling decisions out of incomplete knowledge (every scheduler is provided with some strictly local knowledge), a provably non-optimal choice,
- Make scheduling decisions out of knowledge, explicitly built via appropriate algorithms, so that every scheduler eventually knows what every other scheduler may know,
- Make scheduling decisions out of common knowledge, explicitly built via appropriate algorithms, so that every scheduler knows that every scheduler knows that … (how to build common knowledge in distributed systems is sketched out in Section 5).

As is well known, optimal, i.e. necessary and sufficient, FC's are of exponential complexity. For example, in the ORECA study [10], optimal FC's involve factorials in the number of processes, process activation sporadicity intervals, and scheduler activation periods. This translates into huge execution times for an FD Oracle. An operational requirement – not shown in Figure 1 – is an upper bound set on the execution time of the FD Oracle, used whenever a mission is being prepared. Given the tightness of that bound, we had to establish (sufficient) FC's of weak polynomial complexity. Moreover, these FC's were specified in (max, +) algebra, so as to reduce further the execution time of the FD Oracle. Of course, similar choices, analyses and proofs are in order whenever TU-TimeP appear in <p.CBR>.

### 3.3. TU-TimeP made as simple as TimeP

TU-TimeP can be made equivalent to (classical) TimeP. Consider process k which is associated TUF(k), a non-time varying arbitrary TUF. Off-line, choose a bound U(k), corresponding to completion time D(k), such that (1) TUF(k) $\geq$ U(k) any time before D(k), (2) TUF(k) $\leq$ U(k) any time after D(k). De facto, D(k) is process k's strict latest termination deadline for returning a utility at least equal to U(k). Do this for every process, choosing the U(.)'s so as to achieve some targeted $\alpha$. Clearly, $\Omega$ should be Least-Laxity-First or, preferably, Earliest-Deadline-First (EDF), both of modest complexity. If one cares about overloads, D-Over [12] or similar algorithms should be considered. Timeliness proofs and FC's established for EDF and various CBR's have been published – see [13] for example. They can be entered into PBSE tools.

Similarly, with arbitrary TUF(k), a bound U(k) may correspond to a completion time, denoted d(k), such that (1) TUF(k) $\leq$ U(k) any time before d(k), (2) TUF(k) $\geq$ U(k) any time after d(k). De facto, d(k) is process k's strict earliest termination deadline for returning a utility at least equal to U(k). In which case $\Omega$ is a trivial algorithm, which schedules k's execution to start no earlier than t(k) + d(k) – ExecTime(k), where t(k) is the arrival time of process k's activation request and ExecTime(k) is k's exact execution time (see Section 4.2).

Whenever both deadlines exist for a chosen bound U(k), classical jitter-driven schedulers can be used. Note that EDF or jitter-driven schedulers can be used to process the TUF's shown in Figure 2, since they are concave or quasiconcave functions – any bound U(k) chosen a priori leads to some d(k) or some D(k), or both.

Proceeding as indicated above greatly reduces the complexity involved with proving TU-TimeP, as well as the complexity of on-line schedulers. With many critical applications, it is mandatory to guarantee (with proofs) that some $\alpha$ is always achieved with a CBS, even if that $\alpha$ is not the highest one from a theoretical standpoint, rather than to "expect" some maximization of AU (thanks to some "best effort" scheduling algorithm), without being able to prove a guaranteed lower bound for $\alpha$. Given that proof issues raised with TU-TimeP and specific schedulers have not been very much explored yet, TUF's may hardly be considered for critical applications bound to meet certification obligations. The observations and results briefly presented above help in eliminating this drawback.



IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Note that any set of U(k)'s selected a priori is only one out of (a great) many possible sets. Simplicity has to be traded against an efficiency "penalty", possibly incurred whenever the selected set is sub-optimal (reachable $\alpha$ is higher than the $\alpha$ chosen a priori). However, even less is known yet on *optimality* proofs that would be established for CBS's resting on specific TUF schedulers, even for the non "extreme" TUF's shown in Figure 2. Work sketched out in Sections 3.1 and 3.2, as well as in Section 5, should be conducted (so as to establish optimality proofs).

A final word is in order regarding the issue of "stochastic" vs. "deterministic" analyses, in the presence of uncertainty. Whenever the future – "encapsulated" in <m.CBR>, wcet's, process activation laws, and failure occurrence laws in particular – cannot be known with absolute certainty, two types of analyses can be considered:

• "stochastic" analyses (Bayesian, Markovian, event-driven simulation, etc.), which involve computing a coverage for every probabilistic/statistical model or technique resorted to in the analyses, in addition to the coverage of <m.CBR> (coverage issues arise with probabilistic assumptions – e.g., Poisson arrivals, independence of variables – as well),
• "deterministic" analyses (worst-cases, min-max proofs, etc.), where the only coverage involved is that of <m.CBR>.

The important observation is that uncertainty does not necessarily imply "stochastic" analyses. With some CBR's (problems), simulation – a scientific discipline which may be resorted to for conducting an SDV phase – may suffice. With other CBR's, one must guarantee that a CBS will always "win against" its specified adversary, in which case "deterministic" analyses are mandatory. PBSE does not preclude any of these approaches. PBSE serves to enforce proof obligations, be they proofs of average behaviors and standard deviations, or proofs of worst-case behaviors.

## 4. Brief Overview of TUF/AU Scheduling

### 4.1. Utility of TUF/AU Solutions

In many systems, dynamic service – and thus resource – conflicts and dependencies arise, inevitably, hence waiting queue phenomena. Even with such ultra-simple "design styles" as "time-triggered" (TT) approaches, waiting queues may build up, whenever this is doable according to the adversary encapsulated in <m.CBR>. TT system designers who "rule out" the existence of waiting queues, either violate <m.CBR>, considering an adversary weaker than specified, or simply "displace" waiting queues at the boundaries of a CBS.[2] Unfortunately, in the latter case, the essential issue supposed to be addressed by TT systems, i.e. the meeting of "hard real-time" constraints, is simply ignored, for the reason that "hard real-time" implies establishing an upper bound on sojourn times in waiting queues for every process request. That waiting queues are "shifted" is not equivalent to "no waiting queues".

More generally, regardless of which design "style" is followed, <m.CBR> may be "violated" by the real (future) adversary. Load models (arrival laws) and failure models (failure semantics and density of failure occurrences) are of particular relevance here. No design "style" can help in predicting the future with absolute accuracy. Hence the issue of predicting – along with proofs – the behavior of a CBS when operated beyond its FC's. Whenever the case, some properties specified in <p.CBR> are or may be lost. It is generally required that SafeP should never be jeopardized. Conversely, TimeP may be lost, for some process(es). TUF/AU scheduling serves to dynamically determine which process(es) are immune to violations of <m.CBR>, by dynamically instantiating those schedules which maximize AU (ideally).

TUF/AU concepts match the complexity of many current and future real world applications, such as, e.g., autonomous spacecraft formations, on-line automated finance/stock markets, airline seat reservation (yield management), of current and future computing and networking technology, such as mobile ad hoc wireless systems. Also, they are in line with the work carried out by various communities, such as, e.g., the Distributed Algorithms community, where NP-complete and NP-hard combinatorial problems are examined routinely.

In order to broaden TUF/AU scheduling applicability, it is necessary to address issues raised when combining TU-TimeP with SafeP, LiveP and DepP requirements, the case in particular with applications and systems listed above.

### 4.2. Current TUF/AU Solutions

The state of the art in TUF/AU scheduling [1, 2, 3, 4, 5] is summarized in Figure 3.

***<m.CBR>***
• distributed processors
• process models: sequences and trees, process/processor assignment is unrestricted, inter-process (thread) causal dependencies are known
• a process (thread) can be preempted by another, any time, except when in a critical section, or aborted (unless declared as "non abortable")
• a process (thread) may release resources whenever desired, i.e. before it completes its execution
• ExecTime(k, t), the exact remaining execution time process (thread) k at time t, is known to the scheduler

[2] Except for ultra-simple systems, which do not match the definition of distributed systems, being juxtapositions of more or less independent processors (static functional partitioning).



0-7695-2132-0/04/$17.00 (C) 2004 IEEE

• every instance of a replicated resource is viewed as a distinct resource
• processor failure model: stop
• process (thread) activation models: sporadic, aperiodic
• processor failure occurrence model: aperiodic
• computational model: synchronous.

***<p.CBR>***
• SafeP: mutual exclusion for non-CPU resources
• LiveP: deadlocks or livelocks are eventually resolved
• TU-TimeP: for every process (thread), a TUF and a termination deadline to be met (if missed, process abort); "best-effort" AU maximization
• DepP: processor failure detection within bounded latency.

***Design S***
• every resource access request specifies a HoldTime (after which a requested resource is released)
• processes (threads) may be aborted to maximize AU
• schedulers Ω are specific algorithms
• schedules are recomputed upon the occurrence of (1) every new process (thread) activation request, (2) a new resource allocation request issued by a running process (thread)
• process deadlocks may occur, and deadlock handling is based upon deadlock detection-and-resolution
• resources are released in the reverse order that they are acquired in case of a process (thread) abort
• system-level algorithms (such as deadlock handling) are separate from Ω.

**Figure 3. State of the art in TUF/AU scheduling**

Note that earliest termination deadlines – the d(k)'s, see Section 3.3 – are trivially met with preemptable processes as defined in <m.CBR>. It suffices to defer process termination as appropriate. Note also that with TUF's, one must know ExecTime(k), the exact execution time for process k, rather than wcet(k).

In order to take TUF/UA scheduling a few steps further, it suffices to exhibit designs S that solve CBR's more general, hence more complex, than the CBR shown in Figure 3. An example is presented in Section 5. Regardless of which CBR is being considered, the major challenging *design issue* is that of devising "composite algorithms". Whenever some combination of properties is stated in <p.CBR>, one should avoid examining properties (and deciding on algorithmic solutions) on a individual basis. Algorithmic solutions that can "co-exist" and be "factorized" are highly recommended, for the sake of efficiency – efficiency regarding provability, as well as in terms of run time performance. Let us give an example, with critical updatable variables, in critical distributed CBS's.

Variables are replicated in order to achieve availability (a DepP) despite processor failures. This implies "mutual consistency" – denoted MC, a well known SafeP, which means (informally) that the values taken by any two copies of a given data item should be identical. Therefore, an algorithm ensuring this SafeP should be part of design S, not the case with the S shown in Figure 3. In <m.CBR>, it is stated that every instance of a replicated data item is viewed as a distinct data item. This may mean that MC is not guaranteed – which is consistent with the fact that MC does not appear in <p.CBR>. In case MC would be required, some specific MC algorithm could be "added to" scheduler Ω. This is not efficient. In distributed systems, Ω is bound to be a distributed scheduler. Such a scheduler can be built out of any algorithm solving Consensus (see Section 5). It turns out that MC is achievable also with Consensus algorithms. Moreover, by definition, Consensus algorithms are designed to work correctly in the presence of various failure models. Hence, they serve to achieve some DepP as well. Consequently, in many instances, it is possible to devise a global algorithmic solution having some specific common "skeleton" (e.g., Consensus) capable of enforcing combined SafeP, LiveP, TU-TimeP and DepP at every invocation.

Furthermore, admissible algorithmic solutions for combined SafeP, LiveP and DepP depend strongly on which type of TU-TimeP is specified. Imagine that, in <p.CBR>, "best-effort" is replaced with lower bound α or with some lower bound set for the density of process deadlines to be met. Choosing deadlock-detection-and-resolution (design S in Figure 3) implies that bounds must be analytically established for the following parameters:

• worst-case latency (DL) and utility loss (UL) due to the execution of the selected deadlock detection-and-resolution algorithm, for any scheduled process,
• number of aborts (due to deadlocks) experienced before termination, for any given process,
• number of processes aborted whenever a deadlock is detected (deadlock resolution may lead to more than 1 abort in distributed systems).

Hence the questions: (1) Is it easy to establish such bounds, or would proofs be easier considering another deadlock handling strategy?, (2) DL (resp., UL) having an impact on bounds R(k) (resp., U(k)), which is the deadlock handling strategy that minimizes the DL's and the UL's?

## 5. TUF/AU Scheduling and Distributed Fault-Tolerant Computing

Results (algorithms, proofs of properties, optimal complexity bounds, etc.) established in such areas as, e.g., Distributed Algorithms, Concurrency Control, Dependable Computing, can be used to address issues raised with

"complex" CBR's and CBS's. A reasonably complex and generic CBR has been chosen for the sake of illustration (see Figure 4). Given <m.CBR>, waiting queues may build up, denoted W(p) for processor p, each maintained and serviced by a local scheduler, denoted Ω(p) for processor p. A waiting queue contains names of processes waiting to be activated, of processes waiting to get a resource allocated, of suspended processes (execution to be resumed).

***<m.CBR>***

- distributed processors, some possibly replicated
- application process models: graphs, some possibly replicated, assignment over processors is unrestricted, wcet's and inter-process causal dependencies are known
- processes read/write shared persistent data, data may be replicated
- set of processes, set of shared data items, are open-ended
- processor failure models: stop, omission, timing
- process activation models: sporadic, aperiodic
- processor failure occurrence model: aperiodic
- computational model: any model proven to meet coverage requirement Cov.

***<p.CBR>***

- SafeP: for some processes (typed Z), process serializability and exactly-once semantics (process atomicity, process rollbacks are not allowed)
- LiveP: every process whose activation has been requested eventually terminates
- TU-TimeP: for every process, a TUF and time bounds on "write intervals"; $\alpha$ = targeted global AU ratio
- DepP: processor failure detection within bounded latency, mutual consistency (MC) for multicopied data, uniform consensus, uniform atomic commit
- Cov = $1 - \pi$, $\pi$ = probability that any of the properties may be violated, $\pi < 1.10^{-\beta}$/hour, $\beta > 7$ (critical application).

**Figure 4. A generic CBR**

Any process may write multiple shared data items (e.g., actuators). Having time bounds on "write intervals" means that every time interval between two consecutive writes (triggered by some process) must be no greater than a specified upper bound. Hence, arbitrary process preemption is forbidden. Note also that "best effort" TUF schedulers and properties are ruled out, because of the $\alpha$ requirement. Due to space limitation, we can only sketch out the "composite" algorithmic solution (proofs that the resulting design S solves the generic CBR are omitted).

Serializability means that whenever any two (arbitrarily distributed/replicated) type Z processes A and B conflict at two resources $R_i$ and $R_j$, it must be that either A precedes B at $R_i$ and $R_j$, or B precedes A at $R_i$ and $R_j$. In our case, the only solutions that are admissible are those based upon conflict avoidance [6]. Indeed, process rollbacks being prohibited, deadlock prevention based upon detecting resource conflicts or deadlock-detection-and-resolution are ruled out. Hence, any distributed scheduler Ω capable of dynamically enforcing a system-wide unique total ordering of executions for A and B would achieve specified SafeP and LiveP at once (deadlocks are avoided). This is very similar to Atomic Broadcast or Consensus, where two messages broadcast concurrently by any two processes are delivered in the same order at every processor, despite failures. In fact, Uniform Consensus (UC) is a good "skeleton" for a provably correct overall algorithmic solution. With UC, contrary to Consensus, processors that are about to fail are bound to behave as correct processors – only UC is of interest for real CBS's.

MC holds with UC (see Section 4.2), and UC solves uniform atomic commit also. One can build UC algorithms atop unreliable failure detectors (FD's) [14]. FD's achieve "eventual processor failure detection" only. However, it has been shown how to build Fast FD's in synchronous systems [15]. With Fast FD's, processor failure detection is performed within optimal worst-case lower time bounds. Hence, every DepP holds true. UC, an integral part of Ω, builds common knowledge (see Section 3.2) as follows. Whenever a new type Z process/request must be scheduled at processor p, Ω(p) broadcasts the names (and attributes) of type Z processes contained in W(p), and records the names (and attributes) received from other processors, in order to build an image of the system-wide waiting queue of type Z processes. Then, Ω(p) computes a schedule for the merged set (denoted Sched(p)), and invokes UC with Sched(p) as a proposal. The outcome of UC is a system-wide unique decision: only one of the proposed Sched(.)'s is applied by the processors, despite concurrency and failures. Which enforces TU-TimeP and type Z processes serializability at once. As for $\alpha$, see Section 3.

Cov raises the issue of which computational model should be considered for *specifying* S. The coverage of a computational model is the probability or likelihood that none of its intrinsic timing assumptions can be violated at run time. As is well known, for any given <m.CBR>, and for similar "performance" or "efficiency" figures achieved by a CBS, the coverage of computational models can only increase when moving away from pure synchrony, getting closer to pure asynchrony [9, 16]. Too often, this coverage issue is ignored by proponents of synchronous or time-triggered semantics. It does not help much to claim or to prove such and such properties when claims or proofs rest on extraordinary timing assumptions which have a poor coverage vis-à-vis the real CBS adversary and technology.

Given that high values of $\beta$ need be considered, pure asynchrony "augmented" with FD semantics [14] is a safe choice. Indeed, Strong FD's have been shown to be the



0-7695-2132-0/04/$17.00 (C) 2004 IEEE

weakest semantics for circumventing well known impossibility results established for pure asynchrony. Moreover, it has been shown how to build Perfect FD's (Strong FD's a fortiori) in time-free partially synchronous models [17]. To the best of our knowledge, the resulting model is, among all implementable computational models, the one closest to pure asynchrony identified so far. Hence, this model has the highest coverage (achievable so far). Furthermore, purely asynchronous algorithms – which, being time-free, do not raise any coverage issue regarding postulated timings – may be employed in this model. Hence the choice of a purely asynchronous UC algorithmic "skeleton". UC algorithms built atop Fast FD's are the fastest. Algorithms described in [15, 18], which are Fast UC algorithms, are recommended. How to make use of asynchronous algorithms in real-time systems, while proving TimeP (or TU-TimeP), is explained in [16] – see the "design immersion" or "late binding" principle.

## 6. Conclusion

We have shown how to achieve combined SafeP, LiveP, TU-TimeP and DepP in distributed systems of possibly replicated processors and data, while maximizing AU system-wide. The more difficult issues raised with how to fulfill proof obligations for such combined properties have been addressed, using a powerful and novel methodological approach called Proof-Based System Engineering, which has been briefly presented. Also, we have shown how TU-TimeP reduce to TimeP, easing the proof obligations. That TUF scheduling can be used safely in a large number of computer-based systems should be of interest to many users and designers of critical and non critical applications.

## 7. References


[1] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," *Proc. IEEE Real-Time Systems Symposium*, Dec. 1985, 112-122.

[2] C. D. Locke, *Best-Effort Decision Making for Real-Time Scheduling,,* Ph.D. Thesis, CMU-CS-86-134, Department of Computer Science, Carnegie Mellon University, May 1986.

[3] R. K. Clark, *Scheduling Dependent Real-Time Activites,* Ph.D. Thesis, CMU-CS-90-155, Department of Computer Science, Carnegie Mellon University, Aug. 1990.

[4] E.D. Jensen, "Application QoS-Based Time-Critical Automated Resource Management in Battle Management Systems", *Proc. IEEE Workshop on Object Oriented Real-Time Dependable Systems (WORDS)*, Oct. 2003, 8 p.

[5] P. Li, B. Ravindran, H. Wu, and E. D. Jensen, "A Utility Accrual Scheduling Algorithm for Real-Time Activities with Mutual Exclusion Resource Constraints", *IEEE Transactions on Computers*, submitted Aug. 2003.

[6] P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Pub., 1987, 370 p.

[7] G. Le Lann, "An Analysis of the Ariane 5 Flight 501 Failure – A System Engineering Perspective", *Proc. IEEE Conference on the Engineering of Computer-Based Systems*, March 1997, 339-346. See also "The Failure of Satellite Launcher Ariane 4.5" at http://www.cs.york.ac.uk/hise/hise4/frames9.html
Safety Critical Mailing List, Archived Contributions, Contribution on the Failure of Ariane 5 Flight 501.

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

[9] G. Le Lann, "Predictability in Critical Systems", invited paper, *Proc. 5th Intl. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag LNCS n° 1486, Sept. 1998, 315-338.

[10] P. Carrère, J.-F. Hermant, G. Le Lann, "In pursuit of Correct Paradigms for Object-Oriented Real-Time Distributed Systems", *Proc. IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, May 1999, 271-279. Declassified reports on the French MoD ORECA project (1995-1997) are available from INRIA (in French).

[11] K. Chen, P. Mühlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function", *Journal of Real-Time Systems*, vol. 10, Kluwer Academic Pub., 1996, 293-312.

[12] G. Koren, D. Shasha, "D-Over: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems", *Proc. IEEE Real-Time Systems Symposium*, Dec. 1992, 290-299.

[13] J.A. Stankovic, M. Spuri, K. Ramamritham, G.C. Buttazzo, *Deadline Scheduling for Real-Time Systems*, Kluwer Academic Pub., 1998, 273 p.

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

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

[16] G. Le Lann, "Asynchrony and Real-Time Dependable Computing", *Proc. IEEE Workshop on Object Oriented Real-Time Dependable Systems (WORDS)*, Jan. 2003, 18-25.

[17] G. Le Lann, U. Schmid, *How to Implement a Time-Free Perfect Failure Detector in Partially Synchronous Systems*, Technical Report 183/1-127, Department of Automation, Vienna University of Technology, Jan. 2003, 19 p.

[18] M.K. Aguilera, G. Le Lann, S. Toueg, "On the Impact of Fast Failure Detectors on Real-Time Fault-Tolerant Systems", *Proc. Intl. Conference on Distributed Computing (DISC)*, Springer-Verlag LNCS n° 2508, Oct. 2002, 354-369.