In Pursuit of Correct Paradigms for Object-Oriented Real-Time Distributed Systems

Patrice Carrère, Jean-François Hermant, Gérard Le Lann
INRIA, Projet Reflecs, BP 105
F-78153 Le Chesnay Cedex, France
{Patrice.Carrere | Jean-Francois.Hermant | Gerard.Le_Lann}@inria.fr

Abstract

Real-time distributed applications based on object-oriented technology raise many complex issues, most of them still open. We elaborate on the reasons why traditional paradigms, found adequate for tackling simple problems, cannot help in addressing these complex issues. A detailed illustration is given with a problem in modular avionics, which has been addressed by resorting to proof-based system engineering, an emerging discipline aimed at coping with real-world problems complexity. Drawing from our experience with partners in industry, we report on why such paradigms and technologies as object-orientation, distributed transactions, distributed architectures, transactional monitors, can be contemplated for the construction of real-time distributed systems, provided that appropriate on-line decision-making algorithms and protocols are resorted to.

1. Introduction

State-of-the-art in real-time computing, although rich, is mostly restricted to solutions aimed at specific centralized application areas. Paradigms and solutions abound for, e.g., low-level repetitive (periodic) computing, controlling predictable phenomena, such as fuel injection in a car engine, or orientation of airplane flaps. Still, we cannot ignore that these are particular subcases of more global application problems, of much higher complexity. Furthermore, object-orientation and other technologies inevitably introduce variability within systems, which renders traditional paradigms inappropriate. Stock markets would be a good example of a complex application problem, that clearly calls for on-line (and fast) decisions, under vastly diverse external conditions. We focus on such object-oriented real-time distributed computing (ORC) problems, drawing from our experience with partners in industry. We have selected a problem in modular avionics to illustrate those paradigms which we believe are needed to correctly capture many real world problems, as well as to correctly address them.

In section 2, we introduce the basics of proof-based system engineering (SE), and show how a proof-based SE method was applied to tackle this modular avionics problem (section 3). In section 4 (resp., section 5), we examine solutions regarding computing (resp., communication) issues raised with this problem.

2. Proof-Based System Engineering

Those essential features of proof-based SE needed for the reading of this paper are briefly presented. See [10] for a more detailed presentation. Proof-based SE addresses, in particular, those three phases that come first in a project life-cycle, namely the problem capture, the system design, and the system dimensioning phases (see figure 1, where symbol $\Rightarrow$ stands for "results in"). Notation $\langle Y \rangle$ (resp. $[Z]$) refers to a specification of a problem Y (resp. a solution Z). Notation $\langle y \rangle$ (resp. $[z]$) refers to the specification of a subset of $\langle Y \rangle$ (resp. $[Z]$) which contains those variables which parameterize problem Y (resp. solution Z). The term "specification" refers to a self-contained set of unambiguous statements - in some human language, in some formalized notation, in some formal language.

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

Problem Capture:

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

System Design:

$$\{\langle X\rangle, \langle x\rangle\} \left| \begin{array}{l} \Rightarrow [S] \\ \Rightarrow [s] \\ \Rightarrow [oracle.S] \end{array}\right.$$

System Dimensioning:

$V(\langle x\rangle)$: input to $oracle.S$

$V([s])$: output from $oracle.S$

Figure 1. Phases Covered by Proof-Based System Engineering

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. Notation $\langle x\rangle$ refers 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 $\{\langle m.x\rangle, \langle p.x\rangle\}$.

2.2. The System Design Phase

This phase, conducted by a designer, 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 (generic system S), 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., system architectures, algorithms, modeling), in various theories (e.g., serializability, scheduling, game, complexity), as well as by applying appropriate proof techniques, which depend on those types of problems under consideration.

Activities conducted within a design phase are organized according to a tree structure. A design stage r_i - r being some unique identifier among design tree nodes of level i - consists in solving problem $\langle X(r_i)\rangle$ by deciding on a particular architecture (assuming each architectural module exhibits some local properties) along with algorithms or protocols that encompass this architecture. This leads - in particular - to specify those assumptions (i.e., models) under which the local properties postulated for each architectural module must hold. As a result, problem $\langle X(r_i)\rangle$ is decomposed into independent subproblems of level $i + 1$.

Fulfilling a design correctness proof obligation (design_cpo) guarantees that if every subproblem is correctly solved, then problem $\langle X(r_i)\rangle$ is correctly solved as well by composing the individual solutions. 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 $[S]$.

Design_cpo's They result from class structuring (see end of this section). A solution that solves problem $\langle Y\rangle$ is a correct design solution for problem $\langle X(r_i)\rangle$ if the following two conditions are met:

$$\langle m.Y\rangle \supseteq \langle m.X(r_i)\rangle \text{ and } \langle p.Y\rangle \supseteq \langle p.X(r_i)\rangle.$$

One ends a design tree branch at stage t_k when the specification $[S(t_k)]$ arrived at is deemed implementable, or is known to be implemented via some procurable product (e.g., COTS).

Dimensioning oracle Another output of a design phase is a specification of a system-wide dimensioning oracle - noted $[oracle.S]$ - which includes, in particular, a set of constraints called (system-wide) feasibility conditions (FCs), which 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 - as a tool component - in order to conduct subsequent system dimensioning phase(s).

Quite clearly, composability of design decisions - which, ultimately, translates into composability of physical modules - is one of the essential principles that underlie proof-based SE. A design tree can be constructed following a pure top-down approach. Whenever COTS products or pre-existing implementations must be (re)used, i.e., must be part of final system S, one proceeds bottom-up, specifications of pre-selected products or implementations being some of the design tree leaves.

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 waiting queues, processors speeds, databuses throughputs, number of processors, redundancy degrees. $V([s])$ 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. Given some appropriate metrics (e.g., $\mathbb{R}^+$), one can define dimensioning_cpo's. For example, imagine that dimensioning $V([s])$ guarantees a bound DB on response times for task t, for arrival densities equal to DA. $V([s])$ is correct with respect to t's deadline B (stated in $V(\langle p.x \rangle)$) and t's arrival density A (stated in $V(\langle m.x \rangle)$) if the following holds true: $DA \geq A$ and $DB \leq B$.

Tool component $oracle.S$ either finds a correct $V([s])$, and then declares that there is a quantified S that solves proposed quantified problem $\{\langle X \rangle, V(\langle x \rangle)\}$, or declares that the quantified problem considered is not feasible (with design solution $[S]$).

2.4. Final Comments

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 (as software and/or hardware modules). As mentioned previously, with legacy systems or when COTS products must be (re)used, proof-based SE proceeds bottom-up, some or all of the design tree leaves being known ahead of design time.

Models and properties which are resorted to for conducting these phases, as well as algorithms/protocols, can be organized into classes. Furthermore, it is possible to structure every class after a hierarchy or a partial order. See [13] for an example with the class of failure models. An element that follows another one in a hierarchy or a partial order will be said to be stronger than its predecessor, noted $successor \supseteq predecessor$ [10]. For example, with task models: $graph \supseteq sequence$.

With the event arrival model class, for every event type, one may choose between, e.g, periodic (pm), sporadic (sm), aperiodic (apm), and arbitrary (arm) models, the latter being based on the concept of bounded arrival densities. The unimodal arbitrary model $(uarm)$ is defined as a recurring triple $\{w, n, sp\}$, where w is the size of a sliding time window, n is the maximum number of arrivals (of an event type) within any such window, and sp is a sporadicity interval, i.e. a minimum time separation between two successive arrivals. The multimodal arbitrary model $(marm)$ serves to define a recurring finite and ordered sequence of such triples. Hence: $marm \supseteq uarm \supseteq apm$ and $marm \supseteq uarm \supseteq sm \supseteq pm$.

3. A Problem in Modular Avionics

Our clients in this project [1] were DGA/DRET (French DARPA) and Dassault Aviation [1]. For the sake of conciseness, we will ignore fault-tolerance issues in the sequel. Notation tbd stands for to-be-defined.

Excerpts from application invariant $\langle m.A \rangle$

- Application software (S/W) is modular; dates of creation of application S/W modules (set M of modules) span over many years. Modules can be suppressed or created at will. That is, set M is unbounded.
- Objects, which encapsulate persistent variables that represent current airplane, environment, and system states, are accessed via methods (variables are read and/or written) invoked by application S/W modules.
- Any object may be shared among some unrestricted number of application S/W modules, as well as among (future) system modules and the airplane's environment.
- There should be no restrictions on the programming models used to develop application S/W modules; no restrictions either on which objects can be accessed by a S/W module.
- Distribution of application S/W modules, of shared objects, over S, should be unrestricted.
- List of external event types (e.g., pilot commands, sensors data). Via the updating of an external object (shared by S and its environment), an event occurrence serves to request the activation of one or many application S/W modules which, when executed, produce outputs (object updating), such as responses displayed to pilot, commands applied to actuators.
- Some event types are periodic sensor readings, arrival laws for others (a majority) are tbd.

Excerpts from application invariant $\langle p.A \rangle$

- Many different releases of M, involving any combination of application S/W modules, can be fielded; this should not entail re-designing or re-proving S.
- At all times, states entered by shared objects should consistently mirror the current states of the airplane, of the environment. Object states must also be consistent with mission-dependent variables.
- In order to slash system maintenance and evolution costs, the architecture of S must be modular.
- One should be able to generate any particular release of S "rapidly" (e.g., in less than 30 minutes for a 2-hour long mission).

[1]Project funded by Délégation Générale à l'Armement/DRET, contract 94-395.

Excerpts from problem invariant $\langle m.X \rangle$

- Any set T of transactions [2] drawn from M can be considered; sizes of future T's are tbd.
- Transaction models: finite directed graphs, a node being called an action.
- Run-time model of an action is a sequence. Messages can be generated irrevocably while executing an action.
- Any transaction, if run alone, has a finite and upperly bounded execution time.
- External event type models: Set EV of event types. Mapping of EV onto T is specified via a boolean matrix. Subset EV1: periodic arrivals (values of periods: tbd). Subset EV2: unimodal arbitrary arrivals (values of densities: tbd). Subset EV3: aperiodic arrivals.
- Computational model: synchronous (the conventional model, with Modular Avionics).
- Shared objects: States entered by objects must satisfy client defined invariants (I). Any transaction, if run alone, satisfies (I). Mapping of objects onto T is specified via a boolean matrix.
- Architectural model: a distributed system of modules, no shared memory; highest number of modules is tbd. Mapping of T onto system modules is specified via a boolean matrix.

Excerpts from problem invariant $\langle p.X \rangle$

- Safety: Exactly-once semantics for transactions. No transaction roll-backs. Serializability (every possible run of any given subset of T satisfies (I)).
- Timeliness: Transaction timeliness constraints are strict latest relative termination deadlines.[2] Values of individual deadlines: tbd (values of deadlines ended up ranging from a few milliseconds to one second).
- Complexity $C(oracle)$: pseudo-polynomial in the number of transactions in T.

4. Computing Issues

4.1. Admissible Solutions

Given $\langle X \rangle$, published solutions to ORC problems do not meet the first condition of a design_cpo. Namely, arrival models usually considered are periodic or sporadic, and tasks are modeled as sequences.

4.1.1 Arrival Models

An arrival is a request for launching some activity, of some duration. Consider the case shown in fig. 2(a), whereby a user specifies bounded arrival densities for some particular event type, via the multimodal arbitrary model. Let us assume that worst-case triple is triple 3, i.e. computing backlog (processor load accumulated in a waiting queue) is highest with triple 3. In order to emulate an equivalent recurring worst-case arrival scenario with a periodic model, one would have to specify an arrival frequency of $1/sp_3$ (fig. 2(b)). Therefore, with respect to schedulability, adoption of the periodic model in this case is equivalent to considering artificial processor loads, which leads to - possibly very - pessimistic feasibility conditions (FCs). Indeed, in our case, worst-case triple 3 occurs only once over any time interval of size $w_1 + w_2 + w_3$, while it would occur permanently under equivalent periodic arrival assumptions. An identical conclusion can be drawn whenever unimodal arbitrary models are emulated with periodic models.

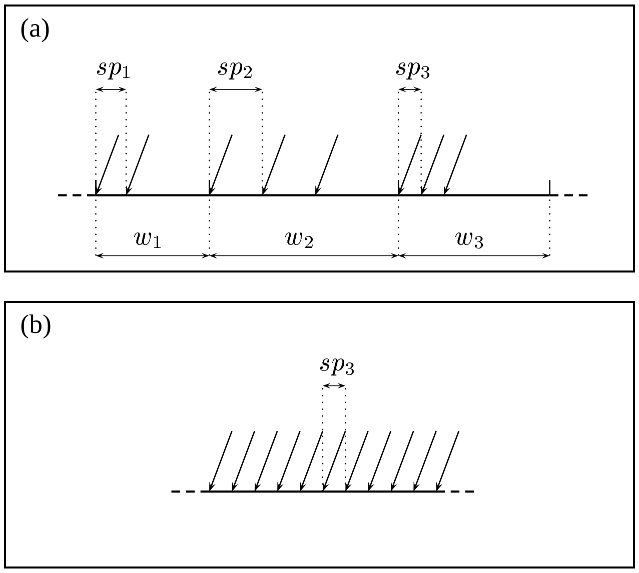

Figure 2. Multimodal Arbitrary Arrivals

In other words, whenever reality is accurately specified via an arbitrary arrival model, as is the case with problem $\langle X \rangle$, there is a price to be paid if one wants to stick to periodic models. Either an (overdimensioned) equivalent periodic model is retained, in which case poor FCs is the penalty incurred, even if one would adopt scheduling algorithms which have been proven optimal (in the restricted class of fixed-priority algorithms), such as HPF (Highest-Priority-First) - priorities being computed a la Rate-Monotonic or Deadline-Monotonic. Or some underdimensioned periodic model is retained, in which case the penalty is that timeliness proofs turn irrelevant. Indeed, proofs that the second condition of a design_cpo is met (no deadline is missed) would be of no value, given that the first condition is violated ($marm \supseteq uarm \supseteq pm$).

[2]If one knows how to solve a "strict deadline" problem, it is straigthforward to solve a "0-jitter" problem.

4.1.2 Task Models

A transaction is modeled as a finite directed graph. Parallelism or dependencies (graph edges) between actions (graph nodes) depend uniquely on the semantics of the application function instantiated as a transaction, i.e. on programmer's decisions. At run time, depending on how objects and transactions are mapped onto the processing modules, a method invocation (object access) may be a strictly local operation or a remote one. How to handle object access concurrency internal to a transaction (invariants (I) must be satisfied by each transaction in isolation from others) is under the responsibility of a programmer. How to handle object access concurrency between transactions (invariants (I) must be satisfied by any run of any set of transactions) is under the responsibility of system designers. Such COTS products as transactional monitors and CORBA-compliant middleware enforce serializability, albeit most of these products/solutions are "blocking" (i.e. deadlocks can occur).

As is the case most often in reality, the execution time of an action is not constant. This time depends on which execution path is followed, which depends on input parameters. Hence, only a worst-case execution time can be predefined for an action. A fortiori, run-time transversal of a transaction depends on input parameters. Messages are generated by actions while executing. Actions have variable run-times and times when messages are generated also depend on input values. Consequently, unimodal or multimodal arbitrary models (rather than periodic or sporadic models) are appropriate for modeling message creation laws. We also offer programmers the possibility of assigning any (relative) deadline to a message. By default, if none is specified, a message deadline is an (analytically established) upper bound on network transit delays (see section 5).

As is well known, transaction serializability is achieved by enforcing a unique and system-wide total ordering ω of conflicting transactions [2]. Consider a processor p, whose waiting queue contains, ranked first, some action of transaction i, noted a_i, and some action of transaction j, noted a_j, ranked $k^{th} (k > 1)$, as per ω. Although a_i is first in waiting queue when processor p becomes available, it might be that a_i cannot execute, as a_i may depend on other actions of transaction i, which are mapped onto other processors, those actions not being completed yet. There are two possible cases. Either a_j does not conflict with any of the actions ranked $1, .., k-1$: a_j may be allocated processor p. Or a_j conflicts with at least one of these actions: a_j cannot run either. If the latter holds for every action of rank $j > 1$ in the waiting queue, p remains idle.

This shows that conventional schedulability analysis based on the concept of "busy periods", taking actual task run-times into consideration only, does not apply here. Processors may be kept idle while waiting queues are not empty, because of (i) dependencies internal to a transaction, (ii) serializability dependencies between transactions.

One can abruptly eliminate cause (i) by pretending that idle times due to intra-transaction dependencies are equivalent to artificial run-times, i.e. artificial processor busy times. These times can be derived from off-line analysis of transaction graphs. Unfortunately, this cannot be done for eliminating cause (ii), i.e. inter-transaction dependencies, because such times depend on which transaction precedes another one in a waiting queue, and this is only run-time decidable. Which explains why we could not use the schedulability results established for task "cohorts" (fixed dependencies). Another difficulty stems from the fact that traditional schedulability analysis assumes task commutativity. By exchanging two sequential tasks in a schedule, a busy period is kept unchanged. This is not the case with transactions. "Shapes" of graphs being arbitrary, exchanging two graphs in a schedule modifies the busy period.

4.1.3 Algorithmic Solutions

The handling of inter-transaction conflicts due to concurrency in accessing objects cannot be based on conflict detection, as this would imply transaction roll-backs so as to solve the deadlock problem, which is not acceptable (see $\langle X \rangle$). Hence, conflicts must be avoided. In addition to serializability, timeliness properties are required. Hence, global time (to be constructed) must be accessible. Global time is used in our solution to enforce conflict avoidance. Given that waiting queues are unavoidable, solutions that work only under the assumption that there are no waiting queues to be scheduled are ruled out. For example, the time-triggered approach [8] is invalid in our case. Two observations are in order. First, from a theoretical viewpoint, "distributed systems without waiting queues" is an oxymoron - and a violation of well known impossibility results[3]. Second, any attempt made at approximating this paradigm has far-reaching implications. Trying to solve $\langle X \rangle$ under a time-triggered approach leads to the artificial constraint that worst-case run-time of any transaction of subset T_p mapped onto processor p must be less than p's "input polling step", i.e. less than ratio p's polling period/number of p's input channels, which is certainly unacceptable with many ORC applications.

Such solutions as HPF with Priority Ceiling (PC) cannot be considered either, given that PC is not usable in a distributed system (assumptions that underlie PC are violations of impossibility results).

[3]Unless (1) one considers it acceptable to discard events - i.e. to lose task activation requests - restricted to be periodic trackings of continuous, predictable, and differentiable functions, (2) worst-case task run-times are assumed to be smaller than the smallest time gap between any two consecutive (accepted) task activation requests, waiting queues do build up.

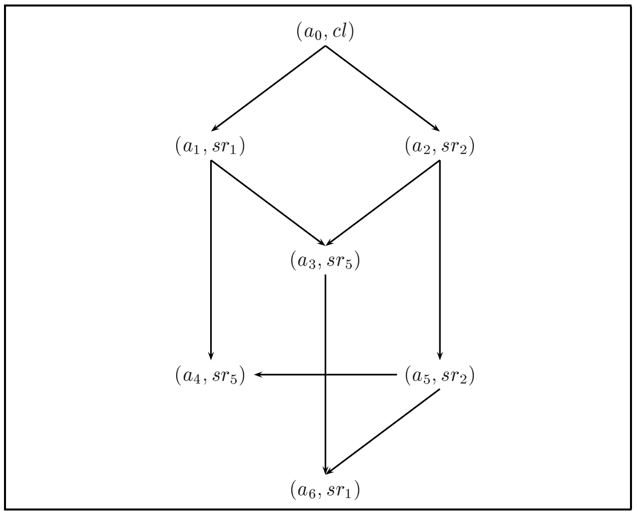

Figure 3. Graph of T

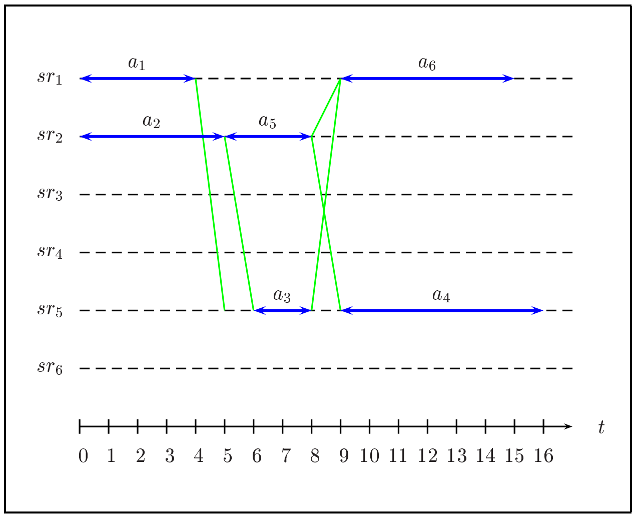

Figure 4. Execution of T

4.2. Sketch of our Solution

The architectural solution is a client-server model. Client (cl) and server (sr) modules are interconnected via a deterministic network module (bounded transmission times). Clients (resp. servers) serve to multiplex sensors (resp. actuators). Root(s) of a transaction (in-node(s) of a graph) is/are mapped onto client(s), while other actions are mapped onto servers. Shared objects are mapped onto servers. Execution of a transaction's actions (a's) is constrained by graph-defined logical synchronization. Fig. 4 shows the execution of transaction T (graph shown in Fig. 3) over a set $\{sr_1, \ldots, sr_6\}$ of servers. In this example, respective upper bounds on execution times of actions $a_1, \ldots, a_6$ are 4, 5, 2, 7, 3, 6 time units, and the upper bound on network transmission time is 1 time unit. All servers are idle at time $t = 0$.

Problem $\langle X \rangle$ contains a scheduling problem which is NP-hard. Hence, theoretically optimal algorithmic solutions are out of question. Would their run-time be accounted for, they turn into non optimal solutions. The algorithmic solution (alg) which we have specified in delivered $[S]$ is a combination of periodic distributed agreement, idling and non-idling (non-idling for urgent transactions[4]), preemptive and non-preemptive, First-In-First-Out, Earliest-Deadline-First, and template-based schedulers (for transactions other than urgent ones). Scheduling decisions are made on-line, periodically, or aperiodically for urgent transactions.

With arbitrary arrival and graph models combined, worst-case scenarios are not necessarily those corresponding to highest arrival densities, contrary to centralized scheduling problems. This observation holds true with the HRTDM problem as well (see section 5). In $[oracle.S]$, one finds those FCs we have established by devising pre-computed schedule templates (not precomputed schedules) and particular constructs on graphs, which have been used to give an appropriate definition of a "busy period". This has permitted to reduce $C(oracle)$ to what was specified. Consequently, our FCs are sufficient, albeit closer to necessary and sufficient FCs than would be achieved with pure FIFO, or pure HPF, or pure EDF.

Worst-case scenarios were formally specified by resorting to matrix calculus in (max, +) algebra [3]. Let VL^+ (resp. VL^-) be the vector of times when servers become idle after (resp. before) the execution of some transaction. Let P be the (max, +) transformation matrix corresponding to a transaction graph. Vector VL^+ is given by $P \otimes VL^-$, $\otimes$ being the (max, +) product.

With the previous example, as we assume that servers are ready to execute T at time $t = 0$, one has $VL^- = [0,0,0,0,0,0]^t$, and $VL^+ = [15,8,0,0,16,0]^t$.

$$VL^+ = \begin{bmatrix} 14 & 15 & -\infty & -\infty & 9 & -\infty \\ -\infty & 8 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ 14 & 16 & -\infty & -\infty & 9 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 15 \\ 8 \\ 0 \\ 0 \\ 16 \\ 0 \end{bmatrix}.$$

Specifications $[S]$ and $[oracle.S]$ have been implemented by Dassault Aviation. The implementation of solution alg was found to be straightforward. The run-time speed-up ratio of $oracle.S$ compared to an event-driven simulator built by Dassault Aviation was found to be in the order of 25.

Furthermore, an event-driven simulator delivers statistically computed distributions (of response times, of sizes of

[4]Those with deadlines in the order of a few milliseconds.

waiting queues), whereas a dimensionning oracle computes upper bounds (of the same variables), which is required in the case of "hard" real-time problems.

5 Communication Issues

For any design stage (see section 2), there is one out of two possible outcomes. Consider $[S(r_k)]$, of design tree level k. If deemed implementable, that specification (of a real (physical) module of S) terminates a design tree branch. If deemed non implementable as such, then $[S(r_k)]$ translates into a level $k+1$ problem - say $\langle X(c_{k+1})\rangle$ - trivially extracted from $[m(r_k)]$, i.e. those models found in $[S(r_k)]$. Let us illustrate the above with the modular avionics problem, noted $\langle X(a_1)\rangle$. As explained above, $[S(a_1)]$ is based - in particular - on a network module, noted node r_1 in the design tree. In $[m(a_1)]$, one finds the following for node r_1 (excerpts):
• message arrival model: multimodal arbitrary (one sequence of triples per message),
• synchronous computational model : network delays range between min and max,
• interval [min, max] comprises all message deadlines.

Given the external event arrivals and the transaction models specified in $\langle m.X(a_1)\rangle$ on the one hand, the algorithmic solution $alg(a_1)$ on the other hand, one derives time windows within which messages can be generated by actions, i.e. the message arrivals model to be found in $[m(a_1)]$. Such derivations, established during design stage 1, may raise non trivial issues, barely addressed so far in scientific publications, one notable exception being the holistic approach [6, 14]. Obviously, this postulated network module is not directly implementable with existing COTS products, nor with documented proprietary products. In fact, $[m(a_1)]$ raises the following level 2 subproblem, say design tree node c_2 (excerpt):
• $\langle m.X(c_2)\rangle$ = multimodal arbitrary message arrivals,
• $\langle p.X(c_2)\rangle$ = no message deadline is missed.

Solving this subproblem entails deciding upon a network architecture, designing a real-time communication protocol (an algorithmic solution) and establishing timeliness proofs, which we have done [5]. However, regarding FCs, in [5], we have considered unimodal arbitrary arrivals, i.e. a weaker problem (in the $\supseteq$ sense), called the HRTDM problem (see below).

Assume for a moment that external event arrivals (requests for activating transactions) are periodic, or that there are ways of enforcing periodic message generation by every action. Unless extraordinary assumptions are made - e.g., absence of waiting queues - we would still have to consider assumption $\langle m.X(c_2)\rangle$, or unimodal arbitrary arrivals to the very least. Indeed, transit times of messages flowing between an application S/W module and a network module are inevitably variable. This is due, in particular, to software (possibly, hardware) layers sitting in between an application layer on the one hand, a network layer on the other hand (e.g., calls to operating systems, to middleware services (e.g., ORB), resource contention handling), as well as to the occurrence of partial failures.

Consequently, even for those messages generated by application tasks that are activated periodically, one must abandon the idea that messages are submitted periodically to a network module. In other words, "real-time" protocols which "work well" under the assumption that messages arrive periodically are not particularly attractive, given that this assumption is unrealistic, except for very specific cases.

5.1 The HRTDM Problem

The HRTDM (Hard Real-Time Distributed Multiaccess) problem arises when selecting a broadcast medium as a network architecture. Sources of messages are actions (of transactions), sensors and actuators. A message is an instantiation of a message type. A finite number of message types is defined for every source.

$\underline{\langle m.HRTDM\rangle}$
• Network model: a distributed broadcast medium, modeled as a ternary variable (idle, busy, collision), accessed via a set ST of stations. Number $|ST|$ of stations is tbd.
• Message sources model: Messages are generated by a set SR of sources. Number $|SR|$ of sources is tbd ($|SR| \geq |ST|$). MSG_{sr} is the set of message types defined for source $sr \in SR$. Number $|MSG_{sr}|$ is tbd.
• Mapping model: The mapping (or multiplexing) of sources (i.e., of sets MSG_{sr}) over stations is unrestricted. In other words, any strict partitioning of SR into $|ST|$ classes can be considered.
• Unimodal arbitrary message arrival model: For every message type msg, triple $\{w(msg), n(msg), sp(msg)\}$ defines msg's bounded arrival density.

$\underline{\langle p.HRTDM\rangle}$
• Safety: Successful transmissions of messages must be mutually exclusive.
• Timeliness (real-time): Message timeliness constraints = strict constant relative deadlines for completing transmission, denoted $d(msg)$ for message type msg.

5.2 Admissible Solutions

With broadcast media, solutions are based either on contention avoidance (CA) or on contention detection-and-resolution (CDR). Representatives of the CA category are explicit token-passing algorithms (e.g., Token Bus, the many Fieldbusses), or implicit token-passing (e.g., STDMA (Static Time Division), TTP [9]), possibly augmented with timers and/or fixed priorities. Representatives of the CDR category are carrier-sense algorithms (e.g., Ethernet, CAN).

It is reasonably obvious that probabilistic CDR protocols, such as Ethernet, cannot be considered for the HRTDM problem. There are deterministic variants of CDR protocols which can be considered, although most of them are sub-optimal. It is reasonably obvious that (explicit, implicit) token-passing is far from being the "best" class of solutions for HRTDM. Indeed, any protocol that would be based on static decisions (i.e. computed off-line) is dominated by protocols based on dynamic (i.e. on-line) decisions. There are worst-case scenarios that are accessible to the "adversary" embodied in $\langle m.HRTDM \rangle$ whereby message deadlines are missed with STDMA or TTP, whereas no deadlines is missed, under the same scenarios, with deterministic CDR protocols.

Think of messages transmitted according to some precomputed ordering. Such an ordering being uncorrelated with message deadlines, transmissions can be arbitrarily close to "Latest-Deadline-First", which maximizes the likelihood that some message deadlines are missed. In fact, this can lead to the poorest achievable FCs. In addition to the above, and compared with explicit token-passing, STDMA or TTP have the well-known drawback of wasting communication bandwidth. The size of each time slot precomputed for every station must be such that it can accommodate the longest message issued by any of the sources multiplexed over that station. Whenever a message shorter than the longest one is transmitted, the communication channel is kept unduly silent. Conversely, the assumption that all messages issued by a given station are of equal length is, first, unrealistic for a vast majority of ORC systems, second, antagonistic with configurability and flexibility requirements.

Hence, only protocols based on deterministic CDR should be considered, given that they generate message orderings arbitrarily close to EDF, which minimizes the number of deadline inversions. Of course, there is a run-time cost incurred with any protocol. Comparing these costs only to tell which protocol is better than another one is flawed. The only way to tell whether a protocol is better than another one, for a precisely specified problem, is by comparing their respective feasibility conditions.

5.3 Sketch of our Solution

Our solution comprises a local scheduling algorithm and a distributed communication protocol. Incoming messages are stored by each source in a waiting queue, which is serviced according to EDF (local algorithm). We have developed a deterministic protocol called CSMA/DDCR (which stands for Carrier Sense Multi Access/Deadline Driven Collision Resolution) [5]. This protocol works exactly like the well-known Ethernet CSMA-CD protocol whenever there is no unresolved collision pending. Collisions are resolved deterministically (via m-ary tree searches) rather than probabilistically (via the BEB algorithm). Collision resolutions, i.e. messages transmissions, are deadline-driven, rather than arbitrarily ordered. A deterministic tie breaking algorithm, called CSMA/DCR [11], is resorted to whenever collisions occur among messages of equivalent (close enough) deadlines. By emulating a distributed Non-Preemptive (NP) EDF scheduling algorithm, CSMA/DDCR happens to be closer to optimality than other solutions, given that centralized NP-EDF is an optimal solution for problems equivalent to the centralized variant of HRTDM, for periodic or sporadic arrival models [4, 7].

Determining whether every message $msg \in MSG$ always meets its transmission deadline $d(msg)$ is a NP-hard problem. Given the requirement on C($oracle$) stated in $\langle p.X \rangle$, only pseudo-polynomial time feasibility conditions (FCs) are acceptable. Schedulability analysis is far from being trivial, given that combination of the unimodal arbitrary model and the dynamic behavior of the CSMA/DDCR protocol raises issues similar to those mentioned in section 4.2.

FCs given in [5] have been perfected, thanks to having established the exact upper bound Ξ on the worst-case time needed to resolve collisions via successive m-ary tree searches, that is:

$$\Xi^{m^h}_{u,v} = v\,\frac{m^{\left\lceil \log_m\left(\frac{m}{v}\left\lfloor\frac{u}{2}\right\rfloor\right)\right\rceil}-1}{m-1} + m\left\lfloor\frac{u}{2}\right\rfloor\left(1+\left\lfloor\log_m\left(\frac{m^h}{\frac{m}{v}\left\lfloor\frac{u}{2}\right\rfloor}\right)\right\rfloor\right) - u,$$

where u, v, h respectively denote the number of messages to be transmitted, the number of tree searches needed, and the depth of the m-ary trees. Fig. 5 shows worst-case times needed to transmit $u \in [2, 50]$ collided messages via $v \in [1, 25]$ successive quaternary tree searches.

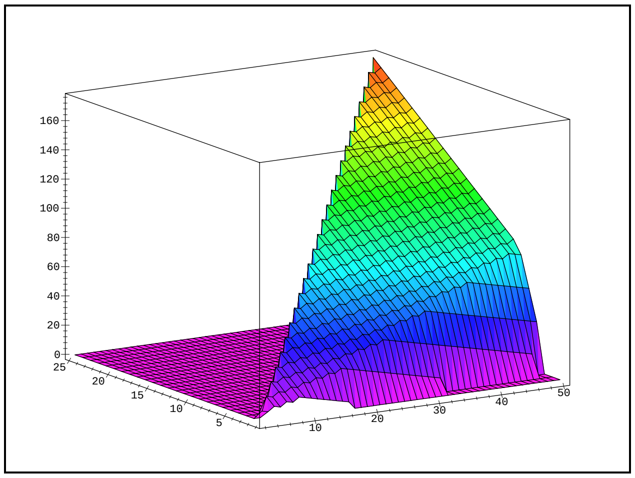

Figure 5. Fonction $\Xi^{4^2}_{u,v}$, $u \in [2, 50]$, $v \in [1, 25]$

6 Conclusions

There are many open issues raised with ORC systems. We believe that it is misleading to claim - without proving - that those traditional paradigms which are adequate for tackling simple problems are appropriate for addressing complex ORC issues, such as those dealt with in this paper. For instance, design solutions based on strong "synchrony" assumptions are very fragile. They "break" abruptly whenever any of their underlying assumptions is violated at runtime. The "predictability" one can expect from such solutions needs careful examination.

We hope this paper contributes usefully to show that such paradigms and technologies as object-orientation, distributed transactions, transactional monitors, ORB-based middleware [12], Ethernet-like networks, can be contemplated for the construction of real-time distributed systems, provided that appropriate on-line decision-making algorithms and protocols are resorted to, such as combinations of distributed agreement and deadline driven scheduling. This has been illustrated with a modular avionics problem, drawn from the international ASAAC program. There is evidence that many real applications raise similarly complex problems.

Coping with real-world problems complexity is one of the goals set to any proof-based SE method. We have reported on how proof-based SE has been applied to address this modular avionics problem. Our method has also been resorted to in other areas (e.g., air traffic control [5], nuclear power plants, satellite launchers, telecommunication networks). A particularly significant reduction in complexity that has been verified - and that can be achieved for similar problems - relates to application software development. It suffices to check the correctness of every program (transaction) in isolation from others to guarantee that the overall application software is correct: correctness is a stable property w.r.t. program composability.

Another appealing feature of proof-based SE is that it permits to verify unambiguously whether some COTS technology is appropriate for a given ORC application problem, without embarking on exponentially complex testing phases.

Being able to design, dimension, build and operate ORC systems satisfactorily is an essential goal set to our community, which has deep theoretical and practical implications.

References

[1] E. Anceaume, L. George, J.-F. Hermant, G. Le Lann, P. Minet, and N. Rivierre. Algorithmique TR/TD/TF ORECA: spécifications des algorithmes et des oracles de faisabilité. Final report (in French), DGA/DRET Contract 94-395, 1995. Restricted Distribution.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Pub., ISBN 0-201-10715-5, 1987.

[3] S. Gaubert. Max plus, methods and applications of (max,+) linear algebra. Research Report 3088, INRIA, Jan. 1997.

[4] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Research Report 2966, INRIA, Sept. 1996.

[5] J.-F. Hermant and G. Le Lann. A protocol and correctness proofs for real-time high-performance broadcast network. In *Proc. of the* 18^{th} *IEEE International Conference on Distributed Computing Systems*, pages 360–369, May 1998.

[6] J.-F. Hermant and N. Spuri. End-to-end response times in real-time distributed systems. In *Proc. of the* 9^{th} *IEEE/ISCA International Conference on Parallel and Distributed Computing Systems*, pages 25–27, Sept. 1996.

[7] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 129–139, Dec. 1991.

[8] H. Kopetz. The time-triggered approach to real-time system design. In *Predictably Dependable Computing Systems*, B. Randell and al. Eds., Springer-Verlag Pub., pages 53–66, 1995.

[9] H. Kopetz and G. Grünsteidl. TTP - A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, Jan. 1994.

[10] G. Le Lann. Proof-based system engineering and embedded systems. In *Lecture Notes in Computer Science 1494*, G. Rozenberg, F. Vaandrager Eds., Springer-Verlag Pub., invited paper, pages 208–248, Oct. 1998.

[11] G. Le Lann and P. Rolin. The 802.3D protocol: A variation on the IEEE 802.3 standard for real-time LANs. Multinational Patent, 1984.

[12] Object Management Group Inc. *The Common Object Request Broker Architecture and Specification; Revision 2.0*. Framingham, USA, July 1995.

[13] D. Powell. Failure mode assumptions and assumption coverage. In *Proc. of the 22*nd *IEEE Fault-Tolerant Computing Symposium*, pages 386–395, July 1992.

[14] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40:117–134, 1994.

[5]The ATR project: Axlog Ingénierie, Dassault Aviation, École Polytechnique/LIX, INRIA, Thomson-Airsys, Université Paris VII/LIAFA, Université de Grenoble/LMC.