Ingénierie système prouvable pour les systèmes informatiques temps réel critiques

Gérard Le Lann
INRIA — Paris-Rocquencourt
Gerard.Le_Lann@inria.fr

Résumé

Les systèmes informatiques temps réel critiques posent de multiples problèmes, qu'il importe de mettre en perspective. Le cycle de vie induit par une méthode d'ingénierie système prouvable pour les systèmes informatiques fournit le cadre recherché. Les principes et les processus d'une telle méthode sont présentés en détail. On montre les liens naturels qui existent entre les considérations de nature scientifique et celles de nature industrielle. Plusieurs problèmes spécifiques du temps réel critique, qui restent ouverts, sont ensuite examinés.

1. Introduction

Les systèmes informatiques temps réel critiques posent des problèmes de nature composite, essentiellement des problèmes relevant à la fois du temps réel strict, de la réplication et/ou la distribution des ressources, de la sûreté de fonctionnement. La démonstration des propriétés exigées de la part de tels systèmes implique un ensemble d'activités complexes, prises individuellement et globalement. La vue d'ensemble nécessaire est fournie par un modèle de cycle de vie, lequel dépend d'une méthode d'ingénierie. Il n'existe pas de contradiction entre « ingénierie » et « sciences exactes », bien au contraire.

On présente tout d'abord les principes fondamentaux de l'ingénierie système à caractère scientifique (prouvable), destinée aux systèmes informatiques en général, aux systèmes informatiques temps réel critiques en particulier. Les principaux problèmes de nature méthodologique, algorithmique, analytique, sont exposés (§2), puis l'ingénierie système prouvable est explorée en détail (§3).

Cette première partie est ensuite illustrée par l'examen de quelques sujets choisis parmi les problèmes ouverts et les analyses a posteriori de systèmes existants (§4).

On utilise les abréviations TR pour « temps réel » et TRC pour « temps réel critique ».

2. Problèmes posés par les systèmes temps réel critiques

2.1. Quelques rappels et définitions élémentaires

a) Système

Un système est un ensemble d'entités qui offre des services à son environnement, lesquels dépendent des propriétés assurées, sous certaines hypothèses, par (1) son architecture, (2) les protocoles et/ou algorithmes gérant les interactions explicites et les conflits (non désirés, par définition) entre entités, (3) les composants technologiques de calcul, de mémorisation et de communication sur lesquels s'exécutent les entités.

Ainsi, par exemple, certaines propriétés inatteignables en architectures distribuées sujettes à rayonnements cosmiques sont atteignables en architectures à mémoire partagée enfouies et protégées.

Donc, on peut, et on doit, raisonner à propos d'un système, spécifier, prouver des propriétés, sans tenir compte des modes de matérialisation des entités (implantations logicielle, microélectronique, optronique, biochips, etc.). Il importe donc de ne pas se laisser abuser par des appellations courantes comme « système à logiciel prépondérant ». Même pour ces systèmes, le logiciel ne peut être la seule problématique posée, loin s'en faut. Que peut-on espérer d'un ensemble de modules logiciels prouvés corrects s'exécutant dans un système défaillant, car non prouvé correct ?

On utilisera le terme « processus » pour désigner les entités exécutables, en ne posant aucune hypothèse relativement à leurs éventuelles implantations.

b) Temps réel

L'appellation « temps réel » caractérise l'ensemble (1) des problèmes algorithmiques de type « décisions en-ligne », (2) des problèmes analytiques, posés par la maximisation de fonctions de gain. Dans les cas particuliers où l'hypothèse d'absence de surcharges est justifiée, ces problèmes se ramènent à démontrer la tenue de contraintes temporelles strictes (propriété de ponctualité (« timeliness »)).

Dans tous les cas, il s'agit de problèmes d'analyse et d'optimisation combinatoires. La ponctualité est une propriété plus « dure » que la terminaison (« liveness »).

Moyennant certaines hypothèses liées à la modélisation du temps physique, on peut spécifier une propriété de ponctualité comme une propriété de sûreté logique.[1]

Néanmoins, par définition, les preuves de ponctualité ne peuvent se ramener à des preuves de sûreté logique. A moins d'ignorer les phénomènes de files d'attente (inévitables dans tous les systèmes, sauf cas rudimentaires) et l'obligation de servir ces dernières selon des algorithmes (ordonnanceurs) entièrement déterminés par le type de propriétés de ponctualité exigé. Faire cette impasse permet de ramener le problème de l'expression de bornes supérieures et inférieures de temps de séjour en files d'attente au problème d'expression de bornes supérieures et inférieures de temps d'exécution. Ou bien de confondre « temps réel » et « conditions de faisabilité d'une propriété P en modèle synchrone » (P n'étant pas spécialement connue comme étant une propriété de ponctualité (exclusion mutuelle, par exemple)).

Noter que les problèmes « temps réel » se posent aussi pour les concepteurs de systèmes « non temps réel » qui postulent un modèle de calcul/système synchrone ou partiellement synchrone (cf. plus loin).

c) Criticité

Tout système « critique » est caractérisé par une exigence de fonctionnement correct de probabilité π au moins égale à $1\text{-}10^{-x}$, la valeur de x étant imposée par le monde réel (clients, autorités étatiques). Par exemple, pour la disponibilité, on a x = 7 en contrôle de trafic aérien, x = 9 pour l'informatique embarquée en avionique civile. Il en découle que toute hypothèse ou modèle considéré pour la conception et les preuves doit avoir un taux de couverture au moins égal à π.

Très souvent, il est utile de partitionner l'ensemble de processus en classes de criticité. Des normes internationales fixent les valeurs de x pour chaque classe, dans divers domaines applicatifs. Voir, par exemple, les niveaux de A à E de la norme DO-178B.

Pour les systèmes « non critiques », on a x « petit », mais nécessairement non nul. Les obligations pour ces systèmes sont donc celles décrites dans cet article. Les différences essentielles pour de faibles valeurs de x sont les suivantes :

- certains résultats d'impossibilité ne s'appliquent pas,
- le « dimensionnement » des systèmes est moindre.

d) TRC implique distribution et sûreté de fonctionnement

De nombreux systèmes TRC (civils, de défense) sont distribués par nature. On sait par ailleurs que l'un des meilleurs moyens d'interdire à un système de tenir des contraintes temporelles est de déclencher des défaillances partielles ou/et des attaques empêchant l'activation et/ou la progression des processus.

Donc, TR implique sûreté de fonctionnement (notée SdF), sauf cas particuliers (problèmes d'informatique de faible complexité, ayant des solutions prouvées optimales depuis longtemps). A fortiori, TRC implique SdF.

On sait que la SdF repose obligatoirement sur de la redondance (informationnelle, spatiale, temporelle), éventuellement diversifiée. On sait aussi que la redondance doit obligatoirement être gérée de façon non centralisée, tout élément centralisateur devenant un point de vulnérabilité, négation évidente du but poursuivi. Ainsi, même si un système TRC n'est pas distribué par nature, il l'est nécessairement pour ses concepteurs, vis-à-vis de la redondance.

e) Distribution

La définition de « distribué » est la définition d'origine [B1], [B2], raffinée et utilisée depuis par toute la communauté « Distributed Algorithms » (ACM, IEEE) : pas de connaissance de l'état global, concomitance des événements et des transitions d'état, pas d'atomicité supposée des pas de calcul ou transitions d'état, ceci caractérisant ce que l'on appelle « systèmes ouverts » en méthodes formelles. L'utilisation de « distribué » dans un sens différent peut introduire des confusions. Par exemple, « distribuer » un logiciel monolithique consiste à le fragmenter et à résoudre ensuite un problème de placement (processus sur processeurs). Ceci ne peut être fait qu'à la condition de supposer une connaissance complète de l'état, contradiction évidente de la définition de « distribution ».

On notera au passage que de nombreux formalismes populaires sont fondés sur des hypothèses caractérisant ce que l'on appelle des « systèmes clos » (connaissance de l'état global, pas de concomitance des événements et des transitions d'état, atomicité supposée des pas de calcul ou transitions d'état). Ils ne peuvent donc pas s'appliquer aux systèmes distribués.

f) Modèles de calcul/système, synchronisme, asynchronisme

La même communauté ACM/IEEE a défini depuis le début des années 80 différentes hypothèses de connaissance préalable concernant les délais (transition d'état, pas de calcul, transfert de message, …), comme suit :

- modèle synchrone : délais finis et bornés, bornes connues (le modèle extrême relativement populaire en ingénierie logicielle formelle étant celui où, de plus, on postule que, pour tout délai, la borne inférieure et la borne supérieure sont égales),
- modèle asynchrone : délais finis, non bornés.

On notera que l'attribut « asynchrone » dans le paradigme GALS (« globally asynchrous locally synchronous ») ne correspond pas à cette définition.

Entre ces deux extrémités du spectre, on trouve divers modèles dits partiellement synchrones (bornes connues pour certains délais, bornes ne devenant vraies qu'à

[1] La « sûreté logique » ne doit pas être confondue avec la « sûreté-innocuité » (non dangerosité).

partir d'un temps futur inconnu, oracles permettant d'éviter les attentes infinies, etc.). Si l'on peut prouver une propriété Ω en modèle asynchrone ou partiellement synchrone, alors Ω est vraie en modèle synchrone. L'inverse n'est pas vrai. Le modèle synchrone est donc un cas particulier de tout autre modèle, du modèle asynchrone en particulier.

Voir [B3] pour une introduction aux modèles de système/calcul, et [B4] pour une présentation de 32 modèles partiellement synchrones.

Ces définitions sont « orthogonales » à celles posées depuis environ 25 ans par la communauté « Real-Time Scheduling » (IEEE) pour ce qui concerne les modèles événementiels (périodiques, sporadiques, apériodiques, arbitraires). Utiliser le qualificatif « asynchrone » pour signifier « non périodique » est donc contradictoire avec ces définitions, et source de confusion.

2.2. Quelques difficultés et pièges notoires

La communauté recherche et la communauté industrielle sont confrontées à des problèmes ouverts, parmi lesquels on trouve :

❶ définir correctement et de façon compréhensible par le « monde réel » (donneur d'ordre, utilisateur) le problème informatique posé, « enfoui » dans l'énoncé d'un problème applicatif/opérationnel, le « monde réel » devant pouvoir s'approprier une telle définition,

❷ satisfaire des obligations de preuves en phase de conception et validation, afin d'éviter les implantations (ultérieures), éventuellement prouvées correctes, de solutions incorrectes,

❸ pouvoir quantifier à loisir des variables laissées libres une fois terminée une phase de conception, et en déduire automatiquement une quantification correcte du système correspondant, sans devoir reprendre un travail de conception et validation,

❹ réutiliser des spécifications existantes validées (problèmes, solutions, preuves),

❺ admettre ou rejeter de façon justifiée des produits du commerce (COTS) en tant que « briques » d'un système.

Parmi les « pièges » qu'il importe d'éviter, on trouve :

❻ se donner des modèles ou hypothèses trop « riches » (taux de couverture inférieurs à π),

❼ confondre solutions système (architectures, algorithmes, protocoles) et leurs implantations,

❽ ignorer les obligations de preuves en phase de conception et validation, et compter principalement sur des tests avant déploiement pour vérifier que le système livré a un taux de couverture au moins égal à π,

❾ tenter de résoudre des problèmes d'algorithmique système par le biais de solutions de niveau applicatif.

Eclairons ce dernier point. Les algorithmes de niveau système ont pour rôle de réguler les exécutions demandées par les processus de niveau applicatif et d'offrir à ces derniers certains services. Par définition, ces algorithmes sont génériques, ils ne dépendent pas d'une sémantique applicative. Voir, par exemple, les algorithmes d'exclusion mutuelle, de contrôle d'accès concomitants (bases de données distribuées), de consensus distribué. Donc, ces algorithmes ne peuvent pas résider au sein de processus de niveau applicatif. De la même manière, en circulation urbaine, les feux de croisement ou les agents de la circulation ne peuvent résider à l'intérieur des véhicules.

Les systèmes informatiques sont des artefacts. Donc, comme pour tous les domaines connus (génie civil, génie électrique, etc.), une « bonne » méthode d'ingénierie ne peut que reposer sur des fondations scientifiques.

Très précisément, toute solution proposée doit être fondée sur des preuves (existantes ou à construire).[2] D'où l'émergence inévitable des méthodes d'ingénierie système prouvable (ISP) pour les systèmes informatiques.

Il peut être instructif de mettre en correspondance les difficultés et pièges répertoriés ci-dessus avec les phases d'un cycle de vie d'ISP :

- définition des exigences (DE) : ❶, ❸, ❻
- conception système & validation (CSV) : ❷, ❸, ❹, ❺, ❻, ❼, ❽, ❾
- faisabilité & dimensionnement (FD) : ❷, ❸, ❹, ❺
- tests d'intégration (TI) : ❹, ❺, ❽.

3. L'ingénierie système prouvable

Le but ultime d'une méthode d'ISP est de produire des systèmes prouvés corrects. Par « preuve », on entend ici toute démonstration à caractère scientifique, comme, par exemple, preuve par récurrence ou réfutation en logique mathématique, démonstration en théorie des graphes, en théorie des ensembles, etc., ou preuve d'optimalité en analyse combinatoire.

Par « preuve », on n'entend pas les explorations mécanisées exhaustives ou quasi-exhaustives d'espaces d'états (utilisées en « model checking »). D'une part, ces techniques ne consistent pas à établir des preuves (contrairement au « theorem proving »). D'autre part, elles sont inenvisageables avec les systèmes distribués (en tout cas en leur état actuel), étant donné la taille gigantesque des espaces d'états atteignables—ne pas oublier que l'on doit tenir compte, par exemple, de l'existence de défaillances partielles et/ou d'attaques, de phénomènes de files d'attente, d'accès conflictuels à des ressources partagées, d'exécutions concomitantes et entrelacées, ce qui contribue grandement à « l'explosion combinatoire » des espaces d'états atteignables.

La figure 1 montre le cycle de vie induit par une méthode d'ISP.

[2] En génie électromagnétique, on ne demande pas aux ingénieurs de re-démontrer les équations de Maxwell, ils les utilisent.

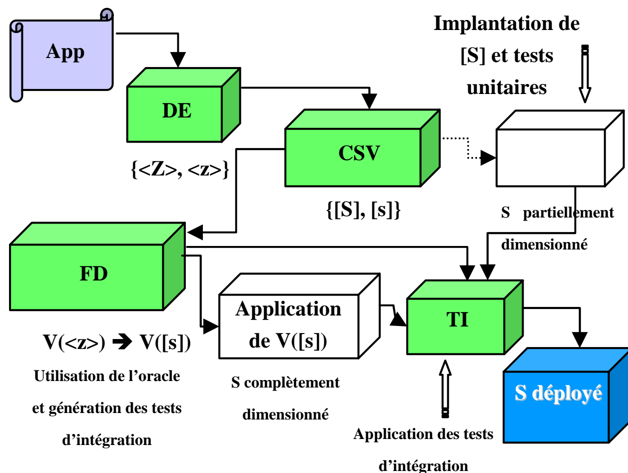

Figure 1. Cycle de vie en ISP

3.1. Phase DE

Pour prouver qu'une spécification [S] de système informatique est correcte pour (« résout ») une spécification <Z> de problème informatique, il est nécessaire de pouvoir établir des spécifications <Z> telles que :

- Z est le problème informatique « enfoui » dans le problème applicatif considéré (App, figure 1),
- ces spécifications ne souffrent pas des travers bien connus de non complétude, ambiguïtés, incohérences, etc., propres aux documents rédigés en langage naturel.

La difficulté majeure tient dans les deux paradoxes suivants :

- soit exprimer <Z> en langage naturel, permettre ainsi son approbation et appropriation par un donneur d'ordre ou client, mais en abandonnant alors l'espoir de faire des preuves,
- soit exprimer <Z> d'emblée en langage formel, ce qui rend impossible son approbation et appropriation par un donneur d'ordre ou client, sachant en outre que l'état de l'art en langages formels ne permet pas d'exprimer toutes les exigences (fonctionnelles, non fonctionnelles) du monde réel.

On ne peut donc qu'exprimer <Z> en langage naturel restreint en première étape d'une phase DE. Dans une spécification <Z> ne peuvent apparaître que des termes (en français, en anglais, etc.) ayant une définition rigoureuse ou formelle dans une discipline scientifique,[3] ce qui élimine les ambiguïtés, tout en offrant la possibilité aux spécificateurs d'exprimer les concepts pertinents pour leurs préoccupations. Par exemple, l'exigence « fichiers toujours cohérents » peut être traduite en des propriétés d'atomicité ou de sérialisabilité associées à un ensemble d'invariants, lesquelles propriétés ont des définitions formelles (théorie des graphes et relations d'ordre partiel ou total).

En simplifiant un peu abusivement, cette première étape a pour sortie une spécification <Z> ni plus ni moins « formelle » qu'un sujet d'examen de seconde année de maîtrise en informatique. Il est établi que l'on peut exiger des démonstrations à partir de telles spécifications. Ensuite, <Z> est « raffinée » pour être formalisée, autant que le permet l'état de l'art scientifique (ce dernier progresse constamment, mais la complexité toujours croissante des exigences et des technologies pose des défis sans cesse renouvelés).

Dans tous les cas réels, il est donc possible de conduire une phase DE de façon rigoureuse, en aboutissant à une spécification <Z> qui est non ambiguë, complète (« self-contained »), cohérente. Noter que recourir à des spécifications en UML ou AADL n'est pratiquement d'aucun secours en la matière, puisque ces dernières ne permettent ni de construire des preuves, ni d'inférer les obligations de preuves à satisfaire.

Certaines des variables qui apparaissent dans une spécification <Z> peuvent être déclarées libres par un donneur d'ordre ou un client, qui souhaite garder la liberté de quantifier ces variables à sa guise dans le futur, par exemple pour différents déploiements (« releases ») d'un système donné. Echéances de terminaison des processus, nombre de processus applicatifs, densités d'occurrence de défaillances partielles, latence maximale de détection d'arrêt de processeur, sont des exemples de telles variables, spécifiées conjointement avec <Z>, spécification notée <z>. La règle qui en découle pour les concepteurs et spécificateurs d'un système S est la suivante : le travail de conception et validation doit être mené de telle sorte qu'il ne soit jamais remis en cause, quelles que soient les quantifications ultérieures des variables de <z>. On verra qu'il en découle l'existence d'une autre spécification, notée [s], qui rassemble les variables libres de la spécification [S]. Nombre de processeurs, nombre de copies multiples de données critiques, période d'activation d'un ordonnanceur, placement de certains processus sur les processeurs, sont des exemples de telles variables. Bien évidemment, [s] est produite en cours de phase de conception et validation. C'est la quantification de <z> qui détermine la quantification de [s].

Un donneur d'ordre ou client peut donc choisir le degré de généricité de <Z>. Il faut choisir un cardinal de <z> élevé si l'on prévoit de nombreuses instanciations « sur mesure » et variées d'un système S, le prix à payer étant que chacune de ces instanciations est sous-optimale. Il faut choisir un cardinal de <z> petit dans le cas contraire, le risque de devoir changer la quantification d'une variable non libre étant alors plus élevé, ce qui entraîne l'obligation de devoir reprendre une phase de conception et validation, travail dont le coût est en général assez élevé.

Une spécification <Z> est constituée de deux sous-ensembles, l'un stipulant les propriétés exigées, l'autre

[3] Les « connecteurs » (articles, prépositions, …) sont admis.

les hypothèses (appelées modèles) sous lesquelles ces propriétés sont exigées.

On a donc <Z> = {<m.Z>, <p.Z>}, où <m.Z> spécifie les modèles et <p.Z> les propriétés.

Les propriétés communément exigées peuvent être regroupées en quatre classes, comme suit :
- sûreté logique (« (logical) safety »), cf. atomicité, sérialisabilité, exclusion mutuelle, etc.,
- vivacité (« liveness »), cf. absence d'interblocage ou quasi-interblocage, terminaison inévitable, etc.,
- ponctualité (« timeliness »), cf. tenue d'échéances ou/et de gigues en l'absence de surcharges, maximisation d'une fonction cumulative de gains en présence de surcharges, etc.,
- confiance ou sûreté de fonctionnement (« dependability »), constituée de deux sous-classes, l'une regroupant les propriétés en présence de failles (erreurs, défaillances partielles) accidentelles (cf. disponibilité, diffusion fiable, accord approché/exact, fiabilité, sûreté-innocuité (« safety »)), l'autre regroupant les propriétés en présence d'attaques intentionnelles (cf. sécurité, confidentialité, authentification, etc.), sachant que certaines propriétés doivent être prouvées en présence à la fois de failles accidentelles et d'attaques intentionnelles.

Les modèles communément considérés sont les suivants :
- calcul/système (synchrone ?, partiellement synchrone ?, asynchrone ?, existence d'horloges ?, etc.),
- données (rémanentes ?, partagées ?, en lecture seulement ?, en lecture/écriture ?, etc.),
- processus (séquence ?, arborescent ?, graphe orienté fini ? pire temps d'exécution (« wcet ») connu ?, etc.),
- défaillances partielles (arrêt définitif ?, omission de message ?, timing ?, valeur erronée ?, arbitraire (byzantin) ?, etc.),
- événementiels pour activation de processus (périodique ?, sporadique ?, apériodique ?, arbitraire à densité bornée ?, etc.),
- occurrences de défaillances partielles (cf. ci-dessus, à l'exclusion du modèle périodique),
- contraintes temporelles (échéance de terminaison au plus tard ?, gigue bornée ?, fonction gain/temps ?, etc.).

Les fonctions « time-value » [B5] ou « time-utility » [B6] sont des exemples de fonction gain/temps.

Une illustration (incomplète) de sortie en première étape de phase DE est donnée ci-dessous, avec un problème de type TRC, noté Φ, étudié à l'occasion d'un contrat.

<m.Φ>

• système distribué de processeurs
• modèle des processus applicatifs: arborescent
• le placement des processus sur les processeurs n'est pas donné
• le pire temps d'exécution de chacun des processus est connu
• les dépendances causales entre processus sont connues
• des variables rémanentes sont partagées en lecture et en écriture par les processus
• certaines variables sont entretenues en copies multiples
• l'ensemble des processus, l'ensemble des données partagées, sont finis et bornés, bornes non données
• communications fiables
• modèles de défaillance des processeurs : arrêt définitif, omission de message, timing
• modèle d'occurrence des défaillances des processeurs : apériodique
• modèles d'arrivées des demandes d'exécution des processus : sporadique, apériodique
• contraintes temporelles des processus typés A : échéances strictes de terminaison ; à chacun des processus non typés A est associée une fonction gain/temps concave ou quasi-concave
• modèle de calcul/système : tout modèle qui satisfait la contrainte π □

<p.Φ>

• sûreté logique : pour les processus typés R, sérialisabilité, roll-backs autorisés ; pour les processus typés C, sérialisabilité, roll-backs interdits ; exclusion mutuelle pour les processus typés E
• vivacité : tout processus dont l'exécution a été demandée finit par se terminer
• ponctualité : pour tout processus typé A, non violation d'échéance; pour les processus non typés A, les gains accumulés sur tout intervalle de temps ne doivent pas être dans un rapport inférieur à α ($\alpha < 1$) à la somme des gains maximaux qui peuvent être accumulés sur l'intervalle considéré
• confiance : détection d'arrêt de processeur en délai fini borné; cohérence mutuelle des données en copies multiples: consensus uniforme, terminaison atomique des processus typés A □

Il peut arriver que l'on soit confronté à un problème sans solution (problème prouvablement infaisable) ou bien sans solution connue. Une méthode d'ISP permet de statuer en cours de phase DE, c'est-à-dire sans perte de temps ni de budgets investis inutilement dans du travail relevant d'une phase CSV, ceci grâce à l'obligation de construire <Z>. On se trouve dans le premier cas lorsqu'un couple de <Z> viole un résultat d'impossibilité (cas vécu en appel d'offre). Dans le second cas, <Z> peut être soumise telle quelle aux experts scientifiques, pour les raisons indiquées précédemment.

3.2. Phase CSV

Une phase de conception système & validation a pour sorties :

- une spécification [S], modulaire et implantable, d'un système informatique S qui est une solution correcte de <Z>,
- les preuves que [S] « résout » <Z>, ou les pointeurs sur ces preuves,
- une spécification [s] qui stipule les variables libres de [S] qui dépendent des variables libres de <z>,

- les conditions de faisabilité du couple {<Z>, [S]},
- la spécification d'un programme (appelé oracle) qui encode les conditions de faisabilité.

Le but des preuves est le suivant :

- établir qu'une architecture dotée de certains protocoles et algorithmes possède des propriétés P*, pour des modèles M*,
- démontrer que P* domine <p.Z> (on prouve qu'existe au moins ce qui est demandé) et que <m.Z> domine M* (on ne suppose pas plus que ce qui est donné), la domination étant définie d'après les relations d'ordre partiel dans les classes de modèles et de propriétés.

Tout comme pour la phase DE, le degré de formalisme ou de formalisation des preuves pour une phase CSV dépend à la fois de la difficulté du problème posé et de l'état de l'art scientifique.

Sauf exception, il n'est pas question d'exiger que les ingénieurs soient astreints à construire les preuves *en cours de projets.* Ce travail incombe aux scientifiques, dont c'est la responsabilité d'anticiper les besoins du monde réel. Une méthode d'ISP doit permettre aux ingénieurs d'utiliser les résultats établis par les scientifiques.

Les obligations de preuves d'une phase CSV sont donc presque toujours satisfaites par « importation » de solutions connues (architecturales, protocolaires, algorithmiques) et des preuves associées. En plus d'être réaliste, cette approche est pertinente, puisque les problèmes système ont un caractère générique très prononcé.

Le processus suivi en phase CSV est de type arborescent. On cesse de développer une branche de l'arbre de conception en s'arrêtant sur une feuille k lorsque deux conditions sont réunies :

- la spécification $[S_k]$ est implantable,
- les modèles postulés pour établir $[S_k]$ et ses preuves ont chacun un taux de couverture au moins égal à π.

Ce processus permet donc de tenir compte des contraintes technologiques (une solution partielle est-elle assez raffinée pour être déclarée implantable ?) et des exigences d'ordre pratique, contractuel ou économique (typiquement, la spécification d'une feuille de l'arbre est-elle satisfaite par un sous-système déjà développé ou par un produit du commerce ?).

In fine, on dispose de [S] sous la forme d'un ensemble de spécifications modulaires, qui peuvent alors être implantées indépendamment les unes des autres, et selon la technologie de réalisation la plus adaptée à chacune d'elles.

On rappelle que c'est en phase CSV que l'on spécifie l'ensemble [s] (variables libres de [S]). Bien évidemment, lorsque [s] n'est pas vide, le système S résultant de ces implantations n'est que partiellement dimensionné.

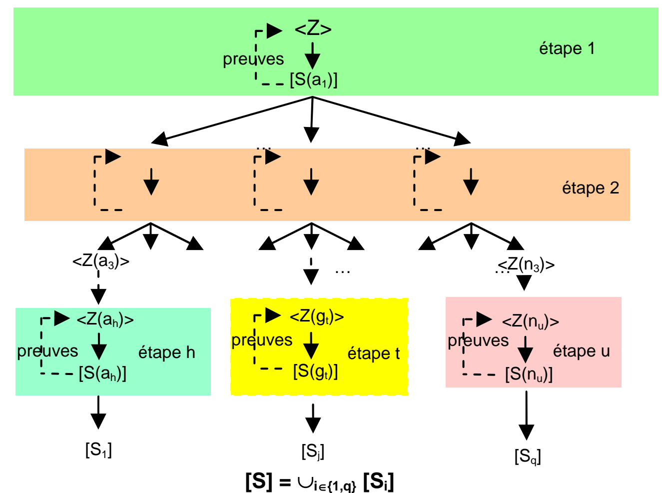

Figure 2. Modélisation d'une phase CSV (arbre de conception)

Une partie des preuves importées ou élaborées au cours de cette phase permet de construire les conditions de faisabilité du couple {<Z>, [S]}. Les conditions de faisabilité sont des systèmes de contraintes analytiques qui expriment les relations existant entre <z> et [s]. Dans le cas des systèmes TRC, ces conditions sont le résultat d'analyses pires cas. Par exemple, on lie ensemble les variables de charges et les échéances de terminaison au plus tard des processus applicatifs. Ou bien on lie ensemble une densité d'occurrence de défaillances partielles et le nombre minimal de rondes pour le consensus distribué.

En cours de phase DE, il est possible de déterminer le type des conditions de faisabilité recherchées (le plus souvent, en moyenne, en pire cas), ainsi que leur optimalité ou complexité maximale admissible (conditions suffisantes, conditions nécessaires et suffisantes). Ces choix—faits par un donneur d'ordre ou un client—sont dictés par des considérations contradictoires (cf. ci-dessous).

3.3. Phase FD

La troisième phase propre à un cycle de vie induit par une méthode d'ISP est la phase de faisabilité & dimensionnement (FD). Le codage des conditions de faisabilité (cf. phase CSV) en vue de leur exploitation donne un programme qu'il est convenu d'appeler un oracle (de faisabilité). C'est sur le couple {<z>, [s]} que « travaille » un oracle.

On rappelle que c'est un donneur d'ordre ou un client qui choisit ou approuve le contenu de <z> en phase DE. C'est grâce à la possibilité ainsi offerte d'assigner ultérieurement des valeurs particulières aux variables de <z> (« late binding ») qu'il est possible de déférer

l'expression de certaines exigences applicatives au-delà de la fin d'une phase CSV, sans invalider cette dernière.

Un oracle fournit les quantifications (suffisantes, nécessaires et suffisantes) des variables système consignées dans [s] (nombre minimal de processeurs, degré de réplication de telle donnée critique, taille minimale pour telle file d'attente, période minimale d'activation de tel algorithme, …) à partir d'une quantification des variables applicatives consignées dans <z> (valeurs des échéances de terminaison au plus tard, niveau de criticité de tel composant, taux maximal d'indisponibilité de tel service, …).

C'est ainsi que l'on évite les défaillances opérationnelles qui sont monnaie courante avec les systèmes sous-dimensionnés, les sous-dimensionnements n'étant pas détectables à l'avance lorsque l'on a recours à des évaluations approximatives (typiquement, des simulations ou des tests, dont les taux de couverture sont rarement estimables avec une précision suffisante—on « valide » seulement une *représentation* d'un système ou un sous-ensemble des états atteignables).

On ne doit pas interpréter de façon réductrice les termes « dimensionnement » ou « quantification ». Bien évidemment, ces termes couvrent l'assignation de valeurs numériques, ordinales, etc. aux variables concernées. Mais il peut également s'agir d'assignations ensemblistes. Ainsi, avec les systèmes d'information (SIC, bases de données distribuées modifiables en-ligne, etc.) ou les systèmes de systèmes, il est préférable de ne pas « figer » trop tôt l'ensemble PA des processus logiciels applicatifs (son existence est stipulée dans <z>). Si, comme c'est presque toujours le cas, on exige au moins les propriétés d'atomicité, cohérence, isolation pour les (futurs) processus applicatifs, il est alors recommandé, en phase CSV, de recourir aux solutions relevant du modèle transactionnel et de la théorie de la sérialisabilité pour les systèmes sans mémoire partagée, ou de la théorie de la linéarisabilité pour les systèmes à mémoire partagée.

Les architectures (redondantes ou simplexes) et les algorithmes distribués qui permettent d'assurer ces propriétés ont été prouvés sans faire aucune hypothèse de connaissance concernant (1) l'ensemble PA, (2) le « contenu » (code, sémantique) in fine des processus considérés.[4] Pour les concepteurs de [S], ces processus sont vus comme des conteneurs à logiciel. Il est donc possible, avant déploiement de chaque nouvelle version ou « release » de S, de « dimensionner » l'ensemble PA, c'est-à-dire de « personnaliser » un système, avec toute composition souhaitable de processus applicatifs (les conteneurs sont « remplis » avec du code dépendant exclusivement d'exigences applicatives). Aucune des phases amont (DE, CSV) ne peut être invalidée par un tel dimensionnement.

[4] La seule hypothèse nécessaire est que chacun des futurs processus est correct pris isolément. Autrement dit, il est ou sera vérifié que chacun de ces processus s'exécute correctement dans un système vide.

Les exigences contradictoires auxquelles est confronté un donneur d'ordre ou un client lorsqu'il doit stipuler le type et la complexité admissible des conditions de faisabilité sont les suivantes. Il est a priori préférable de disposer de conditions nécessaires et suffisantes, car on peut alors dimensionner un système de façon optimale. A contrario, le temps de calcul pris par un oracle peut être jugé « déraisonnable ». Dans de nombreux cas, l'exigence première est le respect d'une borne supérieure des temps de calcul pris par l'oracle.

Très souvent, les bornes utiles en pratique sont petites. Il en résulte que les conditions de faisabilité admissibles sont suffisantes. C'est exactement ce cas qui fut traité lors d'une étude portant sur l'avionique modulaire intégrée pour la Délégation Générale pour l'Armement. Les temps de calcul de l'oracle devaient être « petits » en regard de la durée moyenne d'une mission (2 heures) décidée sur détection de menace.

3.4. Phase TI

En toute rigueur, des obligations de preuves étant satisfaites tout au long du cycle de vie (y compris pour les implantations logicielles des processus, par recours à une méthode d'ingénierie logicielle formelle et à des tests unitaires), une phase de tests d'intégration (TI) n'est pas nécessaire. Néanmoins, étant donné d'une part la complexité des systèmes considérés et, d'autre part, l'impossibilité théorique (cf. le résultat d'incomplétude de Gödel) et pratique (simplement actuelle ou consubstantielle au genre humain ?) de formaliser et de mécaniser entièrement tous les processus d'un cycle de vie, il est « prudent » de procéder à ces tests. A condition d'en réduire la complexité, et de définir les suites de tests d'intégration de façon rationnelle.

La complexité d'une phase TI est considérablement réduite par recours à l'ISP. Cette réduction est obtenue par exploitation des conditions de faisabilité pires cas. Ces conditions déterminent la frontière de l'espace qui contient tous les états atteignables par le système sous test. Par définition, lorsqu'une condition est satisfaite sur cette frontière, alors elle l'est aussi à l'intérieur de cette frontière, c'est-à-dire dans l'espace en question. Il est donc suffisant de tester les points/états sur la frontière, un travail de complexité inférieure de plusieurs ordres de grandeur à celle induite par les méthodes traditionnelles (qui, de facto, « tentent » de tester tous les points/états de l'espace parcourable, une conséquence de l'absence d'obligations de preuves en phase de conception).

Le mode opératoire induit par une méthode d'ISP est le suivant. En sortie de phase CSV, disposant des conditions de faisabilité pires cas, il est possible de générer l'ensemble G des tests permettant de parcourir la frontière déterminée par ces conditions. Une fois qu'un système a été complètement dimensionné (cf. figure 1) et que les tests unitaires sont terminés, on applique les tests contenus dans G.

3.5. Mise en oeuvre de l'ISP

Une méthode d'ISP a été développée à l'INRIA à compter de 1993 (la méthode TRDF). Ses principes et ses phases sont compatibles avec les normes d'IS en vigueur, car cette méthode consiste essentiellement à introduire ce qui est propre aux obligations de preuves dans les cycles de vie faisant l'objet de normes. Ainsi, en simplifiant, pour transformer la norme d'IS ISO/IEC 15288 en norme d'ISP, il « suffit » d'insérer dans la branche « Technical » la phase CSV (manquante) entre les phases « architectural design » et « implementation ». Entre 1995 et 2006, cette méthode a été mise en œuvre dans le cadre de projets, études et contrats relevant de divers domaines, parmi lesquels :

• Avionique Modulaire Intégrée, pour la DGA, avec Dassault Aviation (projet ORECA),

• Refonte du système de contrôle de trafic aérien français, pour la DGAC, avec Thomson Airsys (contrat PHIDIAS),

• Contrôle de trafic aérien (civil) et surveillance aérienne (défense), pour la DGA, le CNRS et le Ministère de la Recherche, avec Thomson Airsys, Dassault Aviation, LIX (projet ATR),

• Refroidissement du cœur des centrales nucléaires, pour l'IPSN,

• Problèmes de coordination autonome dans les grands réseaux de télécommunications, pour France Telecom R&D (projet ALCYON),

• Intergiciels distribués pour satellites, pour l'ESA, avec EADS Astrium (projet A3M),

• Architectures génériques pour systèmes embarqués satellites, pour le CNES (projet AGS),

• Etude et référentiel de mesure de la complexité des systèmes, pour la DGA, avec ATOS/SEMA et Dassault Aviation,

• Etude NCW, pour la DGA, avec DCN et Sagem Défense & Sécurité.

Une présentation de la plupart de ces travaux peut être trouvée dans les références [Ax].

Les principes de la méthode ont également été utilisés pour mener des analyses de causes d'échecs ou de difficultés rencontrés avec des systèmes ou projets opérationnels. Ces analyses ont permis de confirmer les statistiques publiées régulièrement et depuis longtemps par certains des acteurs dominants du domaine (NASA, NIST, …) et par les sociétés spécialisées (Standish Group, …), à savoir que « le logiciel » n'est pas la cause principale des échecs ou difficultés. La grande majorité des échecs ou difficultés résulte de fautes d'ingénierie système (fautes commises dans les phases correspondant aux phases DE ou CSV ou FD d'une méthode d'ISP), c'est-à-dire avant que ne soit produit « le logiciel », ou sans aucune relation avec ce dernier.

Parmi les analyses conduites, on trouve :

- L'abandon d'un projet d'usine « flexible » commandée par une société française à dimension mondiale de produits lactés, après 2,5 années de développement et 2,5 années de procès avec le maître d'œuvre (fautes de conception système, pas de validation),

- La défaillance du vol 501 – Ariane 5, 1996 (faute de définition des exigences),

- Les difficultés du rover Mars Pathfinder, NASA 1997 (fautes de conception système et de dimensionnement),

- Les difficultés du rover Mars Spirit, NASA 2003 (faute de conception système),

- L'échec de la mission de rendez-vous autonome DART, NASA 2005 (faute de conception système).

Une présentation détaillée de certaines analyses peut être trouvée dans les références [Ax].

4. Quelques sujets choisis

Afin d'illustrer les principes de l'ISP, il a fallu faire des choix parmi les très nombreux problèmes ouverts et les analyses a posteriori relatives aux systèmes TRC. Deux éclairages sont tout d'abord donnés sur le C de TRC, l'un concernant les modèles et la contrainte π (§4.1), l'autre concernant les propriétés de SdF globale (§4.2). Puis on examine des problèmes relevant du TR de TRC. On commence par un cas de quasi-échec vécu avec un système TR, pour montrer non seulement que certaines analyses restent superficielles, mais aussi que les principes les plus élémentaires en TR ne sont pas toujours appliqués (§4.3). Puis on explore quelques problèmes ouverts en TR (§4.4).

4.1. Implantation des modèles

Comme on l'a vu, tout module de [S] est prouvé en phase CSV en postulant certains modèles. Pour chacun d'eux, on a deux possibilités :

- soit l'on démontre que le taux de couverture du modèle est meilleur que π ; par exemple, on postule une vitesse de propagation des signaux sur Ethernet égale aux 2/3 de la vitesse de la lumière, hypothèse « garantie » par la Physique,

- soit l'on ne peut établir une telle démonstration ; l'on doit alors fournir une solution (et les preuves) permettant d'implanter le modèle considéré, sous certaines hypothèses (pour lesquelles on a les deux possibilités à nouveau).

Noter que ces obligations doivent également être satisfaites en ingénierie logicielle.

Le cas du modèle synchrone est particulièrement intéressant ici, s'agissant de systèmes TRC. Tout d'abord, rappelons une évidence (preuves triviales) : quelle que soit la réalité opérationnelle et technologique considérée, ce modèle est celui qui a le taux de couverture le plus faible parmi tous les modèles possibles, le modèle asynchrone étant le seul modèle pour lequel la question du taux de couverture ne se pose pas (l'hypothèse est qu'il n'y a pas d'hypothèses sur les délais). En conséquence, l'implantation correcte d'un modèle asynchrone n'implique aucun travail de

conception et preuves, contrairement à ce qui s'impose avec tout autre modèle.

Pour ce qui concerne le modèle synchrone, un avis autorisé a été émis voici plus de 10 ans : "*It is impossible or inefficient to implement the synchronous model in many types of distributed systems.*", N. Lynch [B3]. Pour se convaincre du bien-fondé de cet avis, il suffit de tenter de répondre à la question suivante :

Etant donné un système distribué, dans lequel sont déclenchées des exécutions de processus éventuellement conflictuelles (ressources partagées de façon non explicite), pour chacun selon un modèle événementiel spécifique, chaque processus générant un nombre connu de messages mais à des instants non connus à l'avance, la durée de chaque exécution en l'absence de conflits étant connue, quelles sont les valeurs (expressions analytiques prouvées correctes) des bornes inférieures et supérieures de tout délai (calcul et communication) entre deux processus quelconques ?

Si l'on ne sait pas répondre à cette question, l'on ne sait pas implanter un modèle synchrone dans un système distribué.

Sauf à prouver le contraire, cet avis s'applique a fortiori pour les systèmes TRC. Ainsi, il n'est pas justifié de choisir le modèle synchrone pour la simple raison qu'on le connaît bien ou qu'il simplifie le travail de conception et preuves.

Pour le modèle asynchrone, le problème est qu'il existe de nombreux résultats d'impossibilité lorsque l'on considère l'existence de défaillances. Le plus célèbre est certainement celui de l'impossibilité de consensus (déterministe) dans un modèle asynchrone en présence d'une défaillance [B7].

D'où l'intérêt théorique et pratique fondamental des modèles non situés aux deux extrémités du spectre. Par exemple, si l'on retient le modèle partiellement synchrone obtenu en « enrichissant » le modèle asynchrone avec le détecteur de défaillance parfait $\mathcal{P}$ de Chandra/Toueg [B8], il est impératif d'en donner une implémentation prouvée. C'est ce qui est fait dans [B9] et dans [B10], en considérant pour $\mathcal{P}$ le Θ–Model, un modèle partiellement synchrone fondé sur le ratio Θ(d) de la borne supérieure sur la borne inférieure de tout délai d. L'intérêt de ce modèle est le suivant : pour de nombreux systèmes, il est possible de montrer qu'un ratio Θ(d) n'est pas violé lorsque la borne supérieure du délai d est violée (mauvaise prédiction de cette borne, surcharges, violation de densité d'occurrences de défaillances). Ainsi, pour ces systèmes, une solution prouvée dans le Θ–Model reste correcte dans les scénarios où une solution prouvée en modèle synchrone devient incorrecte.

Cette obligation « d'implanter les modèles » porte aussi, en particulier, sur les modèles de défaillances. Par exemple, si, pour un module de [S], la probabilité d'être en présence d'un comportement byzantin n'est-pas prouvablement inférieure à π, il faut donner une solution système (architecture redondante et algorithmes du type Généraux Byzantins) et les preuves qu'avec une telle solution, le module en question ne peut défaillir selon le modèle byzantin.

Noter également que l'observation de N. Lynch s'applique telle quelle au modèle périodique. En effet, quelle que soit la réalité opérationnelle et technologique considérée, parmi tous les modèles événementiels, le modèle périodique est celui qui a le taux de couverture le plus faible. Il est hautement improbable que les événements postés sur un système distribué par son environnement suivent des lois périodiques. Etant donné l'existence inévitable de phénomènes de files d'attente au sein d'un système (causées par le partage voulu ou accidentel de ressources), une variabilité significative est obligatoirement introduite dans les délais de communications et de traitement. Il est donc encore plus improbable que, par exemple, les lois d'arrivée des messages sur un bus interne ou sur un réseau local soient périodiques.

Les problèmes posés par les implantations prouvées correctes des modèles servant à la conception des systèmes et aux preuves sont peu ou mal examinés. Tout progrès en la matière sera le bienvenu, d'un double point de vue théorique et pratique.

4.2. SdF globale

Il est établi que l'on ne peut prouver des propriétés de confiance qu'à la condition de spécifier :

- les modèles de défaillances considérés,
- les modèles d'occurrence de ces défaillances,
- le ou les modèles de système/calcul considéré(s).

Un treillis pour la classe des modèles de défaillances est donné dans [B11], répertoriant 21 modèles. Le modèle le plus restrictif (le plus facile à gérer) est le modèle « arrêt immédiat et définitif ». Le modèle le plus permissif (le plus difficile à gérer) est le modèle « byzantin » (comportements arbitraires dans les domaines des valeurs et du temps) [B12]. Bien évidemment, les taux de couverture croissent lorsque l'on « s'éloigne » du modèle le plus restrictif, la question du taux de couverture ne se posant pas pour le modèle byzantin.

Les modèles d'occurrence et les preuves de SdF sont de nature stochastique ou « déterministe ». Dans le premier cas, les deux difficultés principales consistent à démontrer que la contrainte π est satisfaite avec les taux de couverture (1) des modèles d'occurrence (par exemple, Poisson), (2) des techniques de modélisation analytique utilisées (par exemple, analyses markoviennes). Dans le second cas, seule la première de ces difficultés est posée (par exemple, « au plus f occurrences pendant une exécution »).

La difficulté de prouver la SdF dans un système TRC tient à l'obligation de couvrir tous les éléments ou niveaux constitutifs d'un système, du niveau logiciels applicatifs et interfaces avec l'environnement (capteurs, par exemple) jusqu'aux composants matériels (par exemple, processeurs et mémoires). Les erreurs qui surviennent à un niveau « inférieur » peuvent créer des défaillances aux niveaux « supérieurs ». D'où l'objectif

de « SdF globale ». Ainsi, le degré croissant de densification des composants matériels est tel que pour des systèmes TRC « grand public », c'est-à-dire concernant chacun d'entre nous à terme (par exemple, réseaux de véhicules autonomes (VANET), systèmes de transport « intelligents » (ITS)), il est devenu nécessaire de concevoir les systèmes embarqués en étendant certains résultats établis pour le spatial.

Les fautes créées dans les mémoires statiques par ionisation due aux rayonnements cosmiques (protons, particules alpha, ions lourds, etc.) font l'objet de travaux importants depuis le milieu des années 70 [B13][5]. Une ionisation peut conduire, par exemple, à une modification du contenu d'une position binaire (« bit flip » de mot mémoire ou de registre) ou à un comportement aléatoire d'une porte logique (« glitch », état métastable)—cf. les SEU (single event upset), les MEU (multiple event upset). Bien que les informations (données, codes exécutables) contenues dans les mémoires et registres soient protégées par des codes correcteurs et/ou détecteurs (Hamming étendu, par exemple), il subsiste des « failles » de protection, principalement les mémoires-caches et les registres dits « externes », notamment ceux utilisés pour les opérations d'entrée-sortie—ce qui concerne au premier chef les protocoles de communication et les algorithmes fondés sur les échanges de messages.

Les problèmes posés sont multiples. Par exemple, comment démontrer qu'un processeur ne se comportera pas éternellement de façon byzantine aux moments où il ne pourra pas être détecté ? Ou bien comment assurer l'immunité d'un processus de « haute » criticité vis-à-vis d'erreurs commises par un processus de « faible » criticité avec lequel il partage des données rémanentes en lecture/écriture ? Quelles sont les relations qui existent dans les réseaux de mobiles entre (1) les déplacements des mobiles, (2) les ruptures de chemins de communication entre deux ou plusieurs mobiles, (3) les modèles de défaillances transitoires et/ou permanentes qui doivent être consignés dans une spécification <m.Z> ?

La raison pour laquelle il ne suffit pas « d'importer » les solutions utilisées dans les systèmes TRC du spatial est triple :

- la propriété de sécurité-innocuité (« safety ») est primordiale pour les systèmes TRC « grand public »,
- les problèmes posés par les intrusions (accidentelles, intentionnelles) au sein des systèmes (pas uniquement vis-à-vis des communications) ne peuvent être ignorés,
- la mobilité est consubstantielle à de nombreux systèmes TRC « grand public », et cette mobilité est sans commune mesure avec celle des véhicules spatiaux (sondes, satellites, rovers).

Il est toujours regrettable de perdre un satellite ou une mission spatiale pour des raisons liées à la SdF. Par contre, il est inenvisageable de faire courir des risques d'accidents aux occupants des véhicules (et ceux situés dans leur environnement) dans les réseaux VANET ou dans les ITS (ou dans d'autres domaines).

Il est donc nécessaire (et urgent) de faire progresser l'état de l'art en matière de SdF globale pour les systèmes TRC.

4.3. De l'utilité fondamentale des analyses d'ordonnançabilité

L'état de l'art en TR *monoprocesseur* pourrait être supposé bien connu. Ce n'est pas toujours le cas, comme l'illustrent les avatars de la mission spatiale Pathfinder/Sojourner.

Quelques jours après l'arrivée de la sonde Pathfinder et du rover Sojourner sur Mars en juillet 1997, Pathfinder devint silencieux. Le centre de contrôle JPL/NASA (Pasadena) envoya un ordre de redémarrage, et Pathfinder put émettre à nouveau, avant de redevenir silencieux. Ce scénario fut répétitif. A chaque fois, il fallait attendre le lendemain pour redémarrer Pathfinder, ce qui compromettait la collecte de données, et à terme la mission, le nombre de jours opérationnels étant limité (batteries de durées limitées).

Par analyse du système « copie » disponible au sol, le JPL finit par découvrir la cause du problème, présentée comme « une inversion de priorités ». Une tâche A de haute priorité (a) et une tâche C de basse priorité (c) partagent un sémaphore X.[6] L'exécution de C peut être suspendue par des tâches B, B', etc., de priorité moyenne b ($c < b < a$) qui n'utilisent pas X. Dans certaines circonstances, le nombre de réquisitions subies par C était tel que C ne se terminait pas « à temps ». Toutes les 125 ms, une tâche N est activée pour initialiser un nouveau cycle (de 125 ms). Aucune tâche n'est censée être en exécution lorsque N est activée. Lorsque N détectait que C était encore en exécution, N déclenchait un « reset » général du système, Pathfinder se mettant alors en attente des ordres du sol. Le moniteur de tâches utilisé (VxWorks de Wind River) offre l'option « héritage de priorité » (HP), qui permet à une tâche de basse priorité (C ici) qui bloque une tâche de plus haute priorité (A ici) d'acquérir sa priorité (a ici). Le booléen HP avait été mis à « faux ». Si HP avait été mis à « vrai », l'exécution de C n'aurait pu être suspendue par les tâches B, B', etc. Et les « resets » n'auraient pas eu lieu. D'où la conclusion « officielle » quant à la cause des difficultés : il s'agit d'un problème de logiciel, lié à une inversion de priorité.

Les « resets » sont une pratique courante dans le spatial. Quand un système ne sait pas « ce qui se passe », il bascule en mode dit « safe » : toute activité est stoppée et il attend des ordres de l'extérieur (sol ou autre).

[5] Des fautes identiques ont été observées dès 1978 avec les mémoires dynamiques au niveau du sol (d'où la nécessité de recourir à des solutions de détection/correction pour, par exemple, les mémoires utilisées dans les véhicules terrestres ou les stimulateurs cardiaques).

[6] On conserve ici le terme de « tâche », couramment employé dans la littérature TR (il n'y a pas de différence entre « tâche » et ce que nous appelons « processus »).

Certes, il existe toujours la possibilité que les véritables conditions opérationnelles soient « pires » que ce qui est spécifié (en début de projet) via les modèles <m.Z>, et dans ce cas le passage en mode « safe » est une transition raisonnable, à condition qu'elle soit exécutée sans violer les propriétés logiques spécifiées via <p.Z>. Mais il arrive aussi que « ne pas savoir ce qui se passe » en opérationnel est la manifestation d'une faute de conception initiale : en phase CSV, on arrête de façon prématurée une analyse devenue trop « compliquée » (car mal conduite). Passer systématiquement en mode « safe » et obéir ensuite aux ordres d'un centre de contrôle peut alors être dangereux. Par chance, la mission Pathfinder a pu être sauvée. Par contre, c'est exactement cela qui a entraîné la perte du satellite européen SPOT 3.

Nous sommes bien évidemment en présence d'un problème TR classique, et d'une analyse (de cause) superficielle. En effet, même avec l'option HP = vrai, il eût pu se faire que des tâches violent leurs échéances de terminaison au plus tard. On sait que la seule façon de prouver que ceci ne peut se produire est de conduire une analyse d'ordonnançabilité, de donner les expressions analytiques des bornes supérieures des temps de réponse, et d'établir les conditions de faisabilité pires cas (exploitables ensuite via un outil, au cours d'une phase FD).[7] Ce qui ne fut pas fait. Pourtant, les conditions de faisabilité pires cas valides pour le système embarqué considéré (monoprocesseur, jeu de tâches fixé, tâches séquentielles, algorithme d'ordonnancement Highest-Priority-First/Priority Inheritance) sont connues et documentées depuis des années.

Premier enseignement : la cause des difficultés n'est pas « l'inversion de priorité », mais le manquement à une obligation de preuves en phase CSV (pas d'analyse d'ordonnançabilité).

Second enseignement : le « logiciel » n'est pas en cause. « Accuser » le logiciel revient à confondre une solution et son utilisation. Le moniteur VxWorks fonctionnait « trop lentement » pour certaines tâches. Comme ce moniteur est instancié en logiciel, il est facile (mais erroné) de confondre « qualité du logiciel » et « choix d'une mauvaise option ». Une analogie pertinente est la suivante : un véhicule avance anormalement lentement, et le conducteur accuse le moteur. Le garagiste révèle la cause véritable du problème : le conducteur n'enclenche que le premier rapport de la boîte de vitesses. Le logiciel qui instancie VxWorks ne souffre d'aucun défaut. Il a été « mal » utilisé. Il ne s'agit donc pas d'une faute d'ingénierie logicielle, mais une faute d'ingénierie système.

Troisième enseignement : l'emploi de priorités fixes est cause de difficultés artificielles. Les avatars de la mission Pathfinder/Sojourner illustrent le vieux débat « priorités fixes ou échéances ». Il est très peu courant qu'un donneur d'ordre ou client exprime (dans <m.Z>) des contraintes temporelles sous la forme d'entiers sans dimension. Transformer des contraintes temporelles en des entiers—les priorités fixes—est faisable sous certaines conditions. Cependant, il est établi qu'une telle transformation ne peut résulter en un système optimal. Un ordonnanceur à priorités fixes (comme Highest-Priority-First) ignore les échéances des tâches en attente d'exécution. Il existe donc de nombreux scénarios faisables avec des ordonnanceurs tel Earliest-Deadline-First qui ne le sont pas avec Highest-Priority-First [B14].

[7] Rappel : cette exigence (analyses d'ordonnançabilité) doit être satisfaite pour tout système, protocole, algorithme, processus, fondé sur des réveils (« timers ») et assurant des services critiques.

4.4. TR distribué et problèmes composites

Par définition (cf. §2.1), l'ordonnancement dans les systèmes TRC est obligatoirement distribué.

a) Nature des problèmes de TR distribué

De façon très simplifiée, le modèle qu'il convient de considérer est le suivant :

- chacun des processeurs dispose d'un ordonnanceur, chargé de servir une file d'attente locale (processus demandés, ou processus suspendus),
- à chaque processus est associée une échéance de terminaison au plus tard,
- certains processus sont exécutables entièrement localement (processus monoprocesseur),
- d'autres processus s'exécutent sur plusieurs processeurs (processus distribués), parce que leur flot d'exécution est réparti sur plusieurs processeurs, ou parce qu'ils ont besoin d'accéder à des ressources distantes,
- les processus monoprocesseur et les processus distribués partagent en lecture/écriture des variables de type rémanent (le cas des données modifiables en-ligne),
- des invariants (ensemble noté C) spécifient les contraintes (dites « de cohérence ») qui lient les valeurs que prennent ces variables « à tout moment ».

Informellement, le problème posé est le suivant : donner un algorithme qui (P1) ordonnance les processus de façon optimale, (P2) de telle sorte que C ne soit jamais violé.

Le problème P2 du respect d'invariants dans le modèle de système considéré a été à l'origine de la théorie de la sérialisabilité [B15]. En bref, une exécution entrelacée de processus est sérialisable si elle est équivalente (pour un observateur extérieur) à une exécution strictement séquentielle. Depuis [B15], de très nombreux algorithmes de contrôle de concomitance (CC) ont été conçus et prouvés assurer la sûreté (C n'est jamais violé). Certains d'entre eux assurent aussi la vivacité, ce qui est indispensable avec les vrais systèmes. Succinctement, le rôle d'un algorithme CC est de construire dynamiquement un ordre total sur les ensembles d'événements « accès aux variables partagées ». En effet, pour que C ne soit pas violé, si deux processus A et B sont en conflit d'accès sur deux variables R_i et R_j (placées sur des processeurs distincts), il faut que soit A précède B sur R_i et R_j, soit B précède A sur R_i et R_j. Dans les systèmes TRC, les processus effectuant des écritures visibles en cours d'exécution (par exemple, commandes d'actionneurs assujetties à

contraintes temporelles), il n'est pas possible de recourir à des algorithmes CC entraînant des roll-backs, le cas des algorithmes à détection d'interblocage et résolution (verrouillage des variables) et des algorithmes à prévention d'interblocage sur détection de conflit (verrouillage des variables ou estampilles). Donc, seuls les algorithmes à évitement de conflits sont éligibles, les relations d'ordre nécessaires étant construites (dynamiquement) avant de commencer les exécutions des processus concernés.

Les algorithmes CC fondés sur les estampilles (plutôt que sur les noms supposés uniques des processeurs, pour raison d'équité) et assurant la vivacité sont donc éligibles. Les ordres d'exécution induits par ces algorithmes sont « par estampilles croissantes » (ou « décroissantes », au choix). La compatibilité avec Earliest-Deadline-First (EDF) est donc parfaite : en présence d'un état donné (file d'attente), les deux algorithmes (CC et EDF) décideront du même ordre d'exécution. Ceci est un exemple d'algorithme « composite », qui « résout » au moins deux problèmes à la fois. Cependant, le problème posé reste ouvert, car P1 n'est pas nécessairement résolu avec EDF, conjecture renforcée par les résultats établis dans [B16].

Le problème (composite) posé appartient en effet à la catégorie des problèmes ouverts, bien que soit omise dans sa présentation une exigence de propriété de SdF, inévitable avec les systèmes TRC.

Il faut donc considérer l'existence de défaillances. Le problème de construire dynamiquement un ordre total sur tout ensemble d'événements, en présence de défaillances, est connu sous le nom de Diffusion Atomique (cf. les actes de la conférence ACM Principles of Distributed Computing) lorsque les événements sont des réceptions de messages diffusés entre processus dans un système distribué. Bien évidemment, l'étape de calcul/communication « diffusion de message » n'est pas supposée atomique ! Un problème proche, le problème Consensus, est celui de choisir (et d'appliquer) une proposition unique parmi plusieurs, qui sont échangées initialement entre processus dans un système distribué, en présence de défaillances. Pour certains éléments du produit cartésien {modèles de système/calcul x modèles de défaillances}, ces deux problèmes sont équivalents. Si chacune des propositions considérées dans le problème Consensus est une relation d'ordre entre messages diffusés, on a résolu Diffusion Atomique quand on a résolu Consensus.

La question de savoir comment résoudre le triple problème composite (TR optimal, C, SdF) est actuellement sans réponse. Une des sources de complexité est la suivante : s'il ne peut exister un algorithme unique « résolvant » les trois problèmes, alors l'algorithme CC et celui assurant la SdF doivent eux-mêmes être ordonnancés (pour exécution) par l'ordonnanceur TR. Comment prouver l'optimalité ?

b) TR et Consensus

Le problème Consensus est posé par la nécessité de gérer les redondances (cf. §2.1). Mais Consensus est également posé directement par le biais d'exigences applicatives comme, par exemple, élection de leader, changement de mode opérationnel, reconfiguration (par exemple, exclusion ou non exclusion d'un processus/processeur), démarrage coordonné d'une opération applicative distribuée (par exemple, modification des positions dans une formation autonome). Donc, en plus d'être un paradigme fondamental en informatique distribuée, Consensus est « omniprésent » dans de nombreux domaines relevant de la thématique TRC (grilles de transport d'énergie, contrôle de trafic aérien, VANET/ITS, etc.).

Consensus a été initialement défini pour des communications fiables et des défaillances de type arrêt pour les processus [B3]. Depuis lors, des résultats ont été établis pour Consensus en considérant des modèles de défaillances plus « durs » (omissions de messages, processus byzantins, par exemple).

Soit un groupe de n processus, groupe connu de tous initialement. Chaque processus propose une valeur initiale choisie librement. Tout processus non défaillant doit choisir de façon irrévocable l'une des valeurs proposées comme valeur de décision (finale). On a résolu Consensus si tous les processus qui décident choisissent la même valeur.

Un algorithme CS résout Consensus pour f défaillances, $0 < f < n$, si lors de chaque exécution de cet algorithme en présence de f défaillances au plus, on a les propriétés suivantes :

- *Terminaison* : tout processus correct finit par décider,
- *Accord* : tous les processus corrects décident identiquement,
- *Validité* : la valeur de décision est l'une des valeurs proposées.

La propriété de terminaison inévitable (« eventual ») est remplacée par la propriété de terminaison ponctuelle suivante dans la version TR :

- *Terminaison ponctuelle* : tout processus correct décide en au plus *u* unités de temps après arrivée de l'événement « demande d'exécution de CS ».

Le problème pertinent pour les « vrais » systèmes est Consensus Uniforme, qui « interdit » aux processus incorrects de décider arbitrairement. La propriété d'accord est remplacée par la propriété suivante :

- *Accord uniforme* : tous les processus décident identiquement.

Il existe de nombreux résultats d'optimalité pour Consensus et Consensus Uniforme, dans divers modèles de système/calcul.

Pour « contourner » le résultat d'impossibilité en modèle asynchrone [B5], il est nécessaire soit de définir des conditions suffisantes temporaires vérifiées pour certaines étapes d'exécution de CS [B17], soit d'enrichir le modèle asynchrone avec des oracles. Les détecteurs de défaillances imparfaits [B8] sont une formalisation axiomatique de tels oracles, bien explorée depuis une douzaine d'années.

Dans [B18], il est montré (1) comment l'on ramène le problème d'implantation des détecteurs de défaillances à un problème d'ordonnancement temps réel de messages dans un réseau, (2) comment construire des détecteurs de défaillances rapides (nécessaire pour le TR), et l'on donne un algorithme de CS rapide. Le tout a été implanté par EADS Astrium [A12].

Cependant, l'état de l'art reste pauvre en résultats concernant Consensus TR. Les problèmes posés sont non triviaux. Mais les défis à relever sont d'un intérêt majeur, tant sur le plan de la recherche que sur le plan des réalisations et des utilisations.

c) TR et mobilité

La problématique TR, l'algorithmique distribuée, la SdF, sont entièrement ré-examinées depuis quelques années dans le cadre des systèmes de mobiles. La plupart des résultats connus jusqu'à récemment sont valides pour (1) des réseaux câblés, assurant la connectivité totale entre processus, (2) des processus non mobiles, (3) des défaillances permanentes, (4) des contraintes temporelles « simples ».

A l'évidence, avec les systèmes de mobiles (ad hoc ou avec infrastructures) qui communiquent par réseaux non filaires, nous sommes en présence de « systèmes ouverts » (cf. §2.1). Le simple fait que la composition d'un système de mobiles soit constamment changeante a de profondes implications.

Prenons le cas des réseaux véhiculaires (VANET). Le problème Consensus TR est posé par un grand nombre de scénarios qui sont hautement critiques, parmi lesquels on trouve :

- insertion dans une file d'un véhicule Y entrant sur une autoroute à haute densité de véhicules circulant à vitesses élevées ; Consensus sert à déterminer le véhicule « gagnant », celui derrière lequel l'espace d'insertion sera créé pour Y, les véhicules suivants étant contraints de ralentir (initialement, chacun des véhicules participants propose d'être le « gagnant »),
- changements de files sur autoroute à haute densité de véhicules circulant à vitesses élevées (changements éventuellement concomitants et conflictuels) ; Consensus (binaire) sert aux véhicules concernés (proches d'un véhicule souhaitant changer de file, ainsi qu'à ce dernier) à décider si le changement peut avoir lieu ou le contraire,
- franchissement d'intersection sans signalisation ou de nuit; Consensus sert à déterminer le véhicule « gagnant », celui qui franchit l'intersection en premier ; ici, on voit que l'on peut utiliser l'équivalence entre Consensus et Diffusion Atomique, afin de désigner non seulement le « gagnant », mais aussi l'ordre des franchissements par tous les véhicules concernés.

Les solutions connues de Consensus sont fondées sur la connaissance *a priori* du groupe de processus concernés. Il est non trivial de « relâcher » cette hypothèse.

Pour le TR, on a plusieurs difficultés qui découlent d'exigences contradictoires avec les modèles de calcul/système qui semblent appropriés. La criticité implique du TR strict. Mais, par exemple, comment prédire une borne supérieure de la durée d'une ronde d'un algorithme de Consensus, sachant que les communications de bout-en-bout entre processus sont sujettes à coupures (fading radio, obstacles, surcharges créant des interférences, etc.) ?

Des travaux récents abordent ces problèmes, mais parfois sous des hypothèses peu réalistes. Par exemple, dans [B19], il est postulé que dans un réseau ad hoc (non filaire) de mobiles :

- les messages sont générés périodiquement,
- il existe un accord implicite entre les mobiles quant à la constitution du réseau (un algorithme « round-robin » fondé sur les identificateurs des mobiles est utilisé).

Les systèmes de mobiles posent de façon emblématique la question (centrale pour les systèmes TRC) de la contrainte π pour les taux de couverture des modèles considérés (cf. §2.1). Pour des raisons de concision, il n'est pas question de développer ici les arguments permettant de se convaincre que TR et modèles non synchrones ne sont pas antinomiques, contrairement à ce que peut le laisser penser une analyse superficielle. On réfère le lecteur intéressé à [B20], où est introduit le concept de « late binding » d'un algorithme asynchrone, qui permet d'en prédire les propriétés de ponctualité.

5. Conclusion

Les systèmes TRC posent un certain nombre de défis majeurs, de nature méthodologique, algorithmique, analytique.

En plus des thèmes examinés ici, il eût fallu en explorer d'autres, pour lesquels subsistent également de nombreux problèmes ouverts. Les formalismes et les techniques de preuves, les systèmes de systèmes, les contraintes induites par l'énergie consommable limitée, la sécurité, sont des exemples.

Le but essentiel de cet article est de mettre en perspective les problèmes posés par les systèmes TRC, et d'en montrer la nature composite. Il est louable de faire progresser la recherche sur des problèmes ouverts spécifiques. Mais la tâche à laquelle doit s'atteler la communauté scientifique, en symbiose avec la communauté industrielle, est aussi de montrer comment résoudre les problèmes composites posés par les vrais systèmes.

Dans cette perspective, l'ISP jouera un rôle fondamental. De multiples domaines applicatifs critiques sont concernés. Il est donc permis de penser qu'une factorisation des travaux pertinents et des connaissances nécessaires se mettra en place progressivement. Puisse cet article y contribuer.

Références des travaux où l'ISP est définie ou utilisée

[A1] INRIA Projet Reflecs, Dassault-Aviation, Etude ORECA, 1995-1997, *rapports pour la DGA*, disponibles sur demande.

[A2] G. Le Lann, "An analysis of the Ariane 5 Flight 501 Failure – A System Engineering Perspective", *Proc. IEEE Intl. Conference on the Engineering of Computer-Based Systems* (Monterey, USA, March 1997), p. 339-346.

[A3] G. Le Lann, "Proof-Based System Engineering and Embedded Systems", invited paper, European School on Embedded Systems (Veldhoven, NL, Nov. 1996), *Lecture Notes in Computer Science n°1494*, Springer-Verlag Pub., Oct. 1998, p. 208-248.

[A4] G. Le Lann, "Models, Proofs and the Engineering of Computer-Based Systems: A Reality Check", Best Paper Award, *Proc. 9th Annual Intl. INCOSE Symposium on "Systems Engineering: Sharing the Future", vol. 4* (Brighton, UK, June 1999), p. 495-502.

[A5] INRIA Projet Reflecs, LIX, Université Paris VII, IMAG, Thomson-Airsys, Axlog, Dassault Aviation, Etude ATR (ISP, contrôle de trafic aérien et avionique modulaire intégrée), 1997-1999, *rapports pour la DGA, le Ministère de la Recherche et le CNRS*, disponibles sur demande.

[A6] G. Le Lann, "The Failure of Satellite Launcher Ariane 4.5", Safety-Critical Forum, 1999, at www.cs.york.ac.uk/hise/safety-critical-archive/1999/0005.html

[A7] 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, p. 339- 346.

[A8] INRIA Projet Reflecs, Université Paris VII, IRISA, Etude AGS, 2001-2003, « Architectures génériques pour systèmes embarqués satellites", *rapports disponibles auprès du CNES*.

[A9] M. Le Roy, P. Gula, J.-C. Fabre, G. Le Lann et E. Bornschlegl, "Novel Generic Middleware Building Blocks for Dependable Modular Avionics Systems", *Proc. ESA Workshop on Spacecraft Data Systems,* (Noordwijk, The Netherlands), 2003.

[A10] EADS Astrium, Axlog, INRIA Projet Reflecs, LAAS, Etude A3M, 2001-2003, « Advanced Avionics Architecture and Modules », *rapports disponibles auprès de l'Agence Spatiale Européenne (ESTEC) et de EADS Astrium.*

[A11] ATOS-Sema, Dassault Aviation, INRIA Projet Reflecs, INRIA Grenoble, Université Paris VI, Université de Montpellier, Etude "Référentiel pour la mesure de la complexité des systèmes et systèmes de systèmes", 2001-2004, *rapports pour la DGA*.

[A12] Honvault C., Le Roy M., Gula P., Fabre J.C., Le Lann G., Bornschlegl E., « Novel Generic Middleware Building Blocks for Dependable Modular Avionics Systems », *Proceedings of the 5th European Conference on Dependable Computing (EDCC-5), Lecture Notes in Computer Science (Springer)*, n°3463, avril 2005, p. 140-153.

[A13] G. Le Lann, « Problèmes de communication et de coordination dans les systèmes spatiaux", papier invité, *actes du Colloque Français sur l'Ingénierie des Protocoles*, Tozeur, Tunisie, oct. 2006, Hermès, 21 p.

Autres références

[B1] G. Le Lann, "Distributed Systems—Towards a Formal Approach", *Proc. Congress of the International Federation for Information Processing (IFIP)*, 1977, North-Holland/Springer, p. 155-160.

[B2] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Com. of the ACM,* vol. 21(7), juillet 1978, p. 558-565.

[B3] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, mars 1996, 907 p.

[B4] D. Dolev, C. Dwork, L. Stockmeyer, « On the Minimal Synchronism Needed for Distributed Consensus », *Journal of the ACM*, vol. 34, n°1, 1987, p. 77-97.

[B5] K. Chen, P. Mühlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function", *Journal of Real-Time Systems*, vol. 10, Kluwer Academic, 1996, p. 293-312.

[B6] B. Ravindran, E. D. Jensen, P. Li, "On recent advances in time/utility function real-time scheduling and resource management", *Proc. 8th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC)*, mai 2005, p. 55-60.

[B7] M.J. Fischer, N.A. Lynch N.A., M.S. Paterson, « Impossibility of Distributed Consensus with One Faulty Process », *Journal of the ACM*, vol. 32, n°2, avril 1985, p. 374-382.

[B8] T.D. Chandra, S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, 43(2), March 1996, p. 225-267.

[B9] 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.

[B10] J.-F. Hermant, J. Widder, "Implementing Reliable Distributed Real-Time Systems with the Theta-Model, *Proc. 9th International Conference on Principles of Distributed Systems (OPODIS 2005), LNCS vol. 3974*, Springer, Pise, Italie, décembre 2005, p. 334-350.

[B11] D. Powell, « Failure Mode Assumptions and Assumption Coverage », *Proceedings of the 22nd IEEE International Symposium on Fault-Tolerant Computing*, juin 1992, p. 386-395.

[B12] L. Lamport, R. Shostak, M. Pease, « The Byzantine Generals Problem », *ACM Transactions on Programming Languages and Systems*, vol. 4, n°3, juillet 1982, p. 382-401.

[B13] T. Karnik T., P. Hazucha P., J. Patel, « Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes », *IEEE Transactions on Dependable and Secure Computing*, vol. 1, n°2, avril-juin 2004, p. 128-143.

[B14] G. Buttazzo, « Rate Monotonic vs. EDF : Judgement Day », *Real-Time Systems Journal (Kluwer)*, vol. 29, n°1, janvier 2005, p. 5-26.

[B15] P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987, 370 p.

[B16] F. Ridouard, P. Richard, F. Cottet, "Negative Results for Scheduling Independent Hard Real-Time Tasks with Self Suspensions", *Proc. 25th IEEE International Real-Time Systems Symposium (RTSS 2004)*, p. 47-56.

[B17] B. Charron-Bost, A. Schiper, « Harmful Dogmas in Fault-Tolerant Distributed Computing », *ACM SIGACT News,* Distributed Computing Column, vol. 38(1), http://www.acm.org/sigactnews/online/, (Whole Number 142), March 2007.

[B18] J.-F. Hermant, G. Le Lann, « Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems», *IEEE Transactions on Computers*, vol. 51, n°8, août 2002, p. 931-944.

[B19] T. Facchinetti, L. Almeida, G. Buttazzo, C. Marchini, "Real-Time Resource Reservation Protocol for Wireless Mobile Ad hoc Networks", *Proc. 25th IEEE International Real-Time Systems Symposium (RTSS 2004)*, p. 382- 391.

[B20] G. Le Lann, "Asynchrony and Real-Time Dependable Computing", *Proc. IEEE Workshop on Object Oriented Real-Time Dependable Systems (WORDS)*, Guadalajara, Mexico, Jan. 2003, p. 18-25.